Welcome! This is the documentation for Python 2.7.13, last updated Dec 17, 2016.

Parts of the documentation:

- What's new in Python 2.7? or all "What's new" documents since 2.0
- Tutorial
  - start here
- Library Reference
  - keep this under your pillow
- Language Reference
  - describes syntax and language elements
- Python Setup and Usage
  - how to use Python on different platforms
- Python HOWTOs
  - in-depth documents on specific topics
- Installing Python Modules
- Index & other sources
- Installing from the Python Package
- Publishing modules for installation by others
- Extending and Embedding
  - tutorial for C/C++ programmers
- Python/C API
  - reference for C/C++ programmers
- FAQs
  - frequently asked questions (with answers!)

Indices and tables:
### Python Module Index

#### _ | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | z

| __ | __builtin__ | The module that provides the built-in namespace. |
| __future__ | Future statement definitions |
| __main__ | The environment where the top-level script is run. |
| _winreg (Windows) | Routines and objects for manipulating the Windows registry. |

<p>| a | abc | Abstract base classes according to PEP 3119. |
| aepack (Mac) | Deprecated: Conversion between Python variables and AppleEvent data containers. |
| aetools (Mac) | Deprecated: Basic support for sending Apple Events |
| aifc | Read and write audio files in AIFF or AIFC format. |
| AL (IRIX) | Deprecated: Constants used with the al module. |
| al (IRIX) | Deprecated: Audio functions on the SGI. |
| anydbm | Generic interface to DBM-style database modules. |
| applesingle (Mac) | Deprecated: Rudimentary |</p>
<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>decoder</td>
<td>Command-line option and argument parsing library.</td>
</tr>
<tr>
<td>argparse</td>
<td>Space efficient arrays of uniformly typed numeric values.</td>
</tr>
<tr>
<td>ast</td>
<td>Abstract Syntax Tree classes and manipulation.</td>
</tr>
<tr>
<td>asynchat</td>
<td>Support for asynchronous command/response protocols.</td>
</tr>
<tr>
<td>asyncore</td>
<td>A base class for developing asynchronous socket handling services.</td>
</tr>
<tr>
<td>atexit</td>
<td>Register and execute cleanup functions.</td>
</tr>
<tr>
<td>audioop</td>
<td>Manipulate raw audio data.</td>
</tr>
<tr>
<td>autoGIL (Mac)</td>
<td><strong>Deprecated</strong>: Global Interpreter Lock handling in event loops.</td>
</tr>
</tbody>
</table>

<p>| b          | |
| base64     | RFC 3548: Base16, Base32, Base64 Data Encodings |
| BaseHTTPServer | Basic HTTP server (base class for SimpleHTTPServer and CGIHTTPServer). |
| Bastion    | <strong>Deprecated</strong>: Providing restricted access to objects. |
| bdb        | Debugger framework. |
| binascii   | Tools for converting between binary and various ASCII-encoded binary representations. |
| binhex     | Encode and decode files in binhex4 format. |</p>
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bisect</strong></td>
<td>Array bisection algorithms for binary searching.</td>
</tr>
<tr>
<td><strong>bsddb</strong></td>
<td>Interface to Berkeley DB database library</td>
</tr>
<tr>
<td><strong>buildtools (Mac)</strong></td>
<td><strong>Deprecated</strong>: Helper module for BuildApplet, BuildApplication and macfreeze.</td>
</tr>
<tr>
<td><strong>bz2</strong></td>
<td>Interface to compression and decompression routines compatible with bzip2.</td>
</tr>
</tbody>
</table>

| **C** | |
| **calendar** | Functions for working with calendars, including some emulation of the Unix cal program. |

**Carbon**

- **Carbon.AE (Mac)** | **Deprecated**: Interface to the Apple Events toolbox. |
- **Carbon.AH (Mac)** | **Deprecated**: Interface to the Apple Help manager. |
- **Carbon.App (Mac)** | **Deprecated**: Interface to the Appearance Manager. |
- **Carbon.Appearance (Mac)** | **Deprecated**: Constant definitions for the interface to the Appearance Manager. |
- **Carbon.CarbonEvents (Mac)** | **Deprecated**: Constants for the interface to the Carbon Event Manager. |
- **Carbon.CarbonEvt (Mac)** | **Deprecated**: Interface to the Carbon Event Manager. |
- **Carbon.CF (Mac)** | **Deprecated**: Interface to the Core Foundation. |
- **Carbon.CG (Mac)** | **Deprecated**: Interface to Core Graphics. |
<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carbon.Cm (Mac)</td>
<td>Deprecated: Interface to the Component Manager.</td>
</tr>
<tr>
<td>Carbon.Components (Mac)</td>
<td>Deprecated: Constants for the interface to the Component Manager.</td>
</tr>
<tr>
<td>Carbon.ControlAccessor (Mac)</td>
<td>Deprecated: Accessor functions for the interface to the Control Manager.</td>
</tr>
<tr>
<td>Carbon.Controls (Mac)</td>
<td>Deprecated: Constants for the interface to the Control Manager.</td>
</tr>
<tr>
<td>Carbon.CoreFounation (Mac)</td>
<td>Deprecated: Constants for the interface to CoreFoundation.</td>
</tr>
<tr>
<td>Carbon.Ctl (Mac)</td>
<td>Deprecated: Interface to the Control Manager.</td>
</tr>
<tr>
<td>Carbon.Dialogs (Mac)</td>
<td>Deprecated: Constants for the interface to the Dialog Manager.</td>
</tr>
<tr>
<td>Carbon.Drag (Mac)</td>
<td>Deprecated: Interface to the Drag and Drop Manager.</td>
</tr>
<tr>
<td>Carbon.Dragconst (Mac)</td>
<td>Deprecated: Constants for the interface to the Drag and Drop Manager.</td>
</tr>
<tr>
<td>Carbon.Events (Mac)</td>
<td>Deprecated: Constants for the interface to the classic Event Manager.</td>
</tr>
<tr>
<td>Carbon.Evt (Mac)</td>
<td>Deprecated: Interface to the classic Event Manager.</td>
</tr>
<tr>
<td>Carbon.Files (Mac)</td>
<td>Deprecated: Constants for the interface to the File Manager.</td>
</tr>
</tbody>
</table>
Carbon.Fm (Mac) Deprecated: Interface to the Font Manager.

Carbon.Folder (Mac) Deprecated: Interface to the Folder Manager.

Carbon.Folders (Mac) Deprecated: Constants for the interface to the Folder Manager.

Carbon.Fonts (Mac) Deprecated: Constants for the interface to the Font Manager.


Carbon.IBCarbon (Mac) Deprecated: Interface to the Carbon InterfaceBuilder support libraries.

Carbon.IBCarbonRuntime (Mac) Deprecated: Constants for the interface to the Carbon InterfaceBuilder support libraries.

Carbon.Icns (Mac) Deprecated: Interface to the Carbon Icon Manager.

Carbon.Icons (Mac) Deprecated: Constants for the interface to the Carbon Icon Manager.


Carbon.LaunchServices (Mac) Deprecated: Constants for the interface to the Carbon Launch Services.

Carbon.List (Mac) Deprecated: Interface to the List Manager.

Carbon.Lists (Mac) Deprecated: Constants for the interface to the List Manager.

Carbon.MacHelp (Mac) Deprecated: Constants for the interface to the Carbon Help Manager.

Carbon.MediaDescr (Mac) Deprecated: Parsers and
generators for Quicktime Media descriptors

Carbon.Menu (Mac)  Deprecated: Interface to the Menu Manager.

Carbon.Menus (Mac)  Deprecated: Constants for the interface to the Menu Manager.

Carbon.Mlte (Mac)  Deprecated: Interface to the MultiLingual Text Editor.

Carbon.OSA (Mac)  Deprecated: Interface to the Carbon OSA Library.

Carbon.OSAconst (Mac)  Deprecated: Constants for the interface to the Carbon OSA Library.

Carbon.Qd (Mac)  Deprecated: Interface to the QuickDraw toolbox.

Carbon.Qdoffs (Mac)  Deprecated: Interface to the QuickDraw Offscreen APIs.

Carbon.QDOffscreen (Mac)  Deprecated: Constants for the interface to the QuickDraw Offscreen APIs.

Carbon.Qt (Mac)  Deprecated: Interface to the QuickTime toolbox.

Carbon.QuickDraw (Mac)  Deprecated: Constants for the interface to the QuickDraw toolbox.

Carbon.QuickTime (Mac)  Deprecated: Constants for the interface to the QuickTime toolbox.


Carbon.Resources (Mac)  Deprecated: Constants for the interface to the Resource Manager and Handles.

Carbon.Scrap (Mac)  Deprecated: The Scrap Manager provides basic...
### Services for Implementing Cut & Paste and Clipboard Operations

- **Carbon.Snd (Mac)**  
  *Deprecated*: Interface to the Sound Manager.

- **Carbon.Sound (Mac)**  
  *Deprecated*: Constants for the interface to the Sound Manager.

- **Carbon.TE (Mac)**  
  *Deprecated*: Interface to TextEdit.

- **Carbon.TextEdit (Mac)**  
  *Deprecated*: Constants for the interface to TextEdit.

- **Carbon.Win (Mac)**  
  *Deprecated*: Interface to the Window Manager.

- **Carbon.Windows (Mac)**  
  *Deprecated*: Constants for the interface to the Window Manager.

- **cd (IRIX)**  
  *Deprecated*: Interface to the CD-ROM on Silicon Graphics systems.

- **cfmfile (Mac)**  
  *Deprecated*: Code Fragment Resource module.

- **cgi**  
  Helpers for running Python scripts via the Common Gateway Interface.

- **CGIHTTPServer**  
  This module provides a request handler for HTTP servers which can run CGI scripts.

- **cgitb**  
  Configurable traceback handler for CGI scripts.

- **chunk**  
  Module to read IFF chunks.

- **cmath**  
  Mathematical functions for complex numbers.

- **cmd**  
  Build line-oriented command interpreters.
Facilities to implement read-eval-print loops.

Encode and decode data and streams.

Compile (possibly incomplete) Python code.

High-performance datatypes

Deprecated: Interface to the standard color selection dialog.

Conversion functions between RGB and other color systems.

Deprecated: Utility functions for running external commands.

Tools for byte-compiling all Python source files in a directory tree.

Deprecated: Python code compiler written in Python.

Configuration file parser.

Utilities for with-statement contexts.

Support for HTTP state management (cookies).

Classes for automatic handling of HTTP cookies.

Shallow and deep copy operations.

Register pickle support functions.

Faster version of pickle, but not subclassable.

The crypt() function used to
<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cStringIO</strong></td>
<td>Faster version of StringIO, but not subclassable.</td>
</tr>
<tr>
<td><strong>csv</strong></td>
<td>Write and read tabular data to and from delimited files.</td>
</tr>
<tr>
<td><strong>ctypes</strong></td>
<td>A foreign function library for Python.</td>
</tr>
<tr>
<td><strong>curses</strong> (Unix)</td>
<td>An interface to the curses library, providing portable terminal handling.</td>
</tr>
<tr>
<td><strong>curses.ascii</strong></td>
<td>Constants and set-membership functions for ASCII characters.</td>
</tr>
<tr>
<td><strong>curses.panel</strong></td>
<td>A panel stack extension that adds depth to curses windows.</td>
</tr>
<tr>
<td><strong>curses.textpad</strong></td>
<td>Emacs-like input editing in a curses window.</td>
</tr>
<tr>
<td><strong>datetime</strong></td>
<td>Basic date and time types.</td>
</tr>
<tr>
<td><strong>dbhash</strong></td>
<td>DBM-style interface to the BSD database library.</td>
</tr>
<tr>
<td><strong>dbm</strong> (Unix)</td>
<td>The standard &quot;database&quot; interface, based on ndbm.</td>
</tr>
<tr>
<td><strong>DEVICE</strong> (IRIX)</td>
<td><strong>Deprecated</strong>: Constants used with the gl module.</td>
</tr>
<tr>
<td><strong>difflib</strong></td>
<td>Helpers for computing differences between objects.</td>
</tr>
<tr>
<td><strong>dircache</strong></td>
<td><strong>Deprecated</strong>: Return directory listing, with cache mechanism.</td>
</tr>
<tr>
<td><strong>dis</strong></td>
<td>Disassembler for Python bytecode.</td>
</tr>
<tr>
<td><strong>distutils</strong></td>
<td></td>
</tr>
</tbody>
</table>
Support for building and installing Python modules into an existing Python installation.

**distutils.archive_util**
Utility functions for creating archive files (tarballs, zip files, ...)

**distutils.bcppcompiler**
**distutils.ccompiler**
Abstract CCompiler class

**distutils.cmd**
This module provides the abstract base class Command. This class is subclassed by the modules in the distutils.command subpackage.

**distutils.command**
This subpackage contains one module for each standard Distutils command.

**distutils.command.bdist**
Build a binary installer for a package

**distutils.command.bdist_dumb**
Build a "dumb" installer - a simple archive of files

**distutils.command.bdist_msi**
Build a binary distribution as a Windows MSI file

**distutils.command.bdist_packager**
Abstract base class for packagers

**distutils.command.bdist_rpm**
Build a binary distribution as a Redhat RPM and SRPM

**distutils.command.bdist_wininst**
Build a Windows installer

**distutils.command.build**
Build all files of a package

**distutils.command.build_clib**
Build any C libraries in a package

**distutils.command.build_ext**
Build any extensions in a package

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

distutils.command.check
Check the metadata of a package

distutils.command.clean
Clean a package build area

distutils.command.config
Perform package configuration

distutils.command.install
Install a package

distutils.command.install_data
Install data files from a package

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

distutils.command.install_lib
Install library files from a package

distutils.command.install_scripts
Install script files from a package

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

distutils.command.sdist
Build a source distribution

distutils.core
The core Distutils functionality

distutils.cygwinccompiler
OS/2 EMX Compiler support

distutils.debug
Provides the debug flag for distutils

distutils.dep_util
Utility functions for simple dependency checking

distutils.dir_util
Utility functions for operating on directories and directory trees

distutils.dist
Provides the Distribution class, which represents the module distribution being built/installed/distributed

distutils.emxccompiler
OS/2 EMX Compiler support

distutils.errors
Provides standard distutils exceptions

distutils.extension
Provides the Extension class,
used to describe C/C++ extension modules in setup scripts

distutils.fancy_getopt
Additional getopt functionality

distutils.file_util
Utility functions for operating on single files

distutils.filelist
The FileList class, used for poking about the file system and building lists of files.

distutils.log
A simple logging mechanism, 282-style

distutils.msvc.compiler
Microsoft Compiler

distutils.spawn
Provides the spawn() function

distutils.sysconfig
Low-level access to configuration information of the Python interpreter.

distutils.text_file
provides the TextFile class, a simple interface to text files

distutils.unixccompiler
UNIX C Compiler

distutils.util
Miscellaneous other utility functions

distutils.version
implements classes that represent module version numbers.

dl (Unix)
**Deprecated:** Call C functions in shared objects.

doctest
Test pieces of code within docstrings.

DocXMLRPCServer
Self-documenting XML-RPC server implementation.

dumbdbm
Portable implementation of the simple DBM interface.

dummy_thread
Drop-in replacement for the thread module.

dummy_threading
Drop-in replacement for the threading module.
<table>
<thead>
<tr>
<th>Module Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EasyDialogs (Mac)</td>
<td>Deprecated: Basic Macintosh dialogs.</td>
</tr>
<tr>
<td>email</td>
<td>Package supporting the parsing, manipulating, and generating email messages, including MIME documents.</td>
</tr>
<tr>
<td>email.charset</td>
<td>Character Sets</td>
</tr>
<tr>
<td>email.encoders</td>
<td>Encoders for email message payloads.</td>
</tr>
<tr>
<td>email.errors</td>
<td>The exception classes used by the email package.</td>
</tr>
<tr>
<td>email.generator</td>
<td>Generate flat text email messages from a message structure.</td>
</tr>
<tr>
<td>email.header</td>
<td>Representing non-ASCII headers</td>
</tr>
<tr>
<td>email.iterators</td>
<td>Iterate over a message object tree.</td>
</tr>
<tr>
<td>email.message</td>
<td>The base class representing email messages.</td>
</tr>
<tr>
<td>email.mime</td>
<td>Build MIME messages.</td>
</tr>
<tr>
<td>email.parser</td>
<td>Parse flat text email messages to produce a message object structure.</td>
</tr>
<tr>
<td>email.utils</td>
<td>Miscellaneous email package utilities.</td>
</tr>
<tr>
<td>encodings</td>
<td></td>
</tr>
<tr>
<td>encodings.idna</td>
<td>Internationalized Domain Names implementation</td>
</tr>
<tr>
<td>encodings.utf_8_sig</td>
<td>UTF-8 codec with BOM signature</td>
</tr>
<tr>
<td>ensurepip</td>
<td>Bootstrapping the <code>pip</code> installer into an existing Python installation or virtual</td>
</tr>
<tr>
<td>Module</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>errno</td>
<td>Standard <code>errno</code> system symbols.</td>
</tr>
<tr>
<td>exceptions</td>
<td>Standard exception classes.</td>
</tr>
<tr>
<td><strong>f</strong></td>
<td><strong>fcnt1 (Unix)</strong>: The <code>fcntl()</code> and <code>ioctl()</code> system calls.</td>
</tr>
<tr>
<td></td>
<td><strong>filecmp</strong>: Compare files efficiently.</td>
</tr>
<tr>
<td></td>
<td><strong>fileinput</strong>: Loop over standard input or a list of files.</td>
</tr>
<tr>
<td><strong>findertools (Mac)</strong></td>
<td>Wrappers around the finder's Apple Events interface.</td>
</tr>
<tr>
<td><strong>FL (IRIX)</strong></td>
<td><strong>Deprecated</strong>: Constants used with the <code>fl</code> module.</td>
</tr>
<tr>
<td><strong>fl (IRIX)</strong></td>
<td><strong>Deprecated</strong>: FORMS library for applications with graphical user interfaces.</td>
</tr>
<tr>
<td><strong>flp (IRIX)</strong></td>
<td><strong>Deprecated</strong>: Functions for loading stored FORMS designs.</td>
</tr>
<tr>
<td><strong>fm (IRIX)</strong></td>
<td><strong>Deprecated</strong>: Font Manager interface for SGI workstations.</td>
</tr>
<tr>
<td><strong>fnmatch</strong></td>
<td>Unix shell style filename pattern matching.</td>
</tr>
<tr>
<td><strong>formatter</strong></td>
<td>Generic output formatter and device interface.</td>
</tr>
<tr>
<td><strong>fpectl1 (Unix)</strong></td>
<td>Provide control for floating point exception handling.</td>
</tr>
<tr>
<td><strong>fpformat</strong></td>
<td><strong>Deprecated</strong>: General floating point formatting functions.</td>
</tr>
<tr>
<td><strong>fractions</strong></td>
<td><strong>Deprecated</strong>: Rational numbers.</td>
</tr>
<tr>
<td><strong>Framework (Mac)</strong></td>
<td><strong>Deprecated</strong>: Interactive application framework.</td>
</tr>
<tr>
<td><strong>ftplib</strong></td>
<td>FTP protocol client (requires environment).</td>
</tr>
</tbody>
</table>
| **functools** | sockets).  
**Higher-order functions and operations on callable objects.** |
| **future_builtins** |
| **g**  |
| **gc** | Interface to the cycle-detecting garbage collector. |
| **gdbm (Unix)** | GNU's reinterpretation of dbm. |
| **gensuitemodule (Mac)** | Create a stub package from an OSA dictionary |
| **getopt** | Portable parser for command line options; support both short and long option names. |
| **getpass** | Portable reading of passwords and retrieval of the userid. |
| **gettext** | Multilingual internationalization services. |
| **gl (IRIX)** | **Deprecated**: Functions from the Silicon Graphics Graphics Library. |
| **GL (IRIX)** | **Deprecated**: Constants used with the gl module. |
| **glob** | Unix shell style pathname pattern expansion. |
| **grp (Unix)** | The group database (getgrnam() and friends). |
| **gzip** | Interfaces for gzip compression and decompression using file objects. |
| **h**  |
| **hashlib** | Secure hash and message digest algorithms. |
heapq

Heap queue algorithm (a.k.a. priority queue).

hmac

Keyed-Hashing for Message Authentication (HMAC) implementation

hotshot

High performance logging profiler, mostly written in C.

hotshot.stats

Statistical analysis for Hotshot

htmlentitydefs

Definitions of HTML general entities.

htmllib

Deprecated: A parser for HTML documents.

HTMLParser

A simple parser that can handle HTML and XHTML.

httpclient

HTTP and HTTPS protocol client (requires sockets).

i

ic (Mac)

Deprecated: Access to the Mac OS X Internet Config.

icopen (Mac)

Deprecated: Internet Config replacement for open().

imageop

Deprecated: Manipulate raw image data.

imaplib

IMAP4 protocol client (requires sockets).

imgfile (IRIX)

Deprecated: Support for SGI imglib files.

imghdr

Determine the type of image contained in a file or byte stream.

imp

Access the implementation of the import statement.

importlib

Convenience wrappers for __import__
<table>
<thead>
<tr>
<th>imputil</th>
<th>Deprecated: Manage and augment the import process.</th>
</tr>
</thead>
<tbody>
<tr>
<td>inspect</td>
<td>Extract information and source code from live objects.</td>
</tr>
<tr>
<td>io</td>
<td>Core tools for working with streams.</td>
</tr>
<tr>
<td>itertools</td>
<td>Functions creating iterators for efficient looping.</td>
</tr>
<tr>
<td>j</td>
<td></td>
</tr>
<tr>
<td>jpeg (IRIX)</td>
<td>Deprecated: Read and write image files in compressed JPEG format.</td>
</tr>
<tr>
<td>json</td>
<td>Encode and decode the JSON format.</td>
</tr>
<tr>
<td>k</td>
<td></td>
</tr>
<tr>
<td>keyword</td>
<td>Test whether a string is a keyword in Python.</td>
</tr>
<tr>
<td>l</td>
<td></td>
</tr>
<tr>
<td>lib2to3</td>
<td>the 2to3 library</td>
</tr>
<tr>
<td>linecache</td>
<td>This module provides random access to individual lines from text files.</td>
</tr>
<tr>
<td>locale</td>
<td>Internationalization services.</td>
</tr>
<tr>
<td>logging</td>
<td>Flexible event logging system for applications.</td>
</tr>
<tr>
<td>logging.config</td>
<td>Configuration of the logging module.</td>
</tr>
<tr>
<td>logging.handlers</td>
<td>Handlers for the logging module.</td>
</tr>
<tr>
<td>m</td>
<td></td>
</tr>
<tr>
<td>macerrors (Mac)</td>
<td>Deprecated: Constant</td>
</tr>
</tbody>
</table>
MacOS (Mac)

Definitions for many Mac OS error codes.

macostools (Mac)

Deprecated: Access to Mac OS-specific interpreter features.

macpath

Deprecated: Convenience routines for file manipulation.

Mac OS 9 path manipulation functions.

macresource (Mac)

Deprecated: Locate script resources.

mailbox

Manipulate mailboxes in various formats.

mailcap

Mailcap file handling.

marshal

Convert Python objects to streams of bytes and back (with different constraints).

Mathematical functions (sin() etc.).

md5

Deprecated: RSA's MD5 message digest algorithm.

mhlib

Deprecated: Manipulate MH mailboxes from Python.

mimetypes

Mapping of filename extensions to MIME types.

MimeWriter

Deprecated: Write MIME format files.

mimify

Deprecated: Mimification and unmimification of mail messages.

MiniAEFrame (Mac)

Support to act as an Open Scripting Architecture (OSA) server ("Apple Events").

mmap

Interface to memory-mapped
files for Unix and Windows.

Find modules used by a script.

Creation of Microsoft Installer files, and CAB files.

Miscellaneous useful routines from the MS VC++ runtime.

Deprecation: Support for reading files which contain distinct parts, such as some MIME data.

Process-based "threading" interface.

API for dealing with sockets.

Dumb wrapper around threading.

Share data between process with shared objects.

Create pools of processes.

Allocate ctypes objects from shared memory.

Deprecation: Lock and queue for mutual exclusion.


Loading of .netrc files.

Deprecation: Interface to the creation of runtime implementation objects.

Interface to Sun's NIS (Yellow Pages) library.

NNTP protocol client (requires sockets).

Numeric abstract base classes
(Complex, Real, Integral, etc.).

<table>
<thead>
<tr>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator</td>
</tr>
<tr>
<td>optparse</td>
</tr>
<tr>
<td>os</td>
</tr>
<tr>
<td>os.path</td>
</tr>
<tr>
<td>ossaudiodev <em>(Linux, FreeBSD)</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>parser</td>
</tr>
<tr>
<td>pdb</td>
</tr>
<tr>
<td>pickle</td>
</tr>
<tr>
<td>pickletools</td>
</tr>
<tr>
<td>pipes <em>(Unix)</em></td>
</tr>
<tr>
<td>PixMapWrapper <em>(Mac)</em></td>
</tr>
<tr>
<td>pkgutil</td>
</tr>
<tr>
<td>platform</td>
</tr>
<tr>
<td>plistlib</td>
</tr>
<tr>
<td>popen2</td>
</tr>
<tr>
<td>Module</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td><strong>poplib</strong></td>
</tr>
<tr>
<td><strong>posix (Unix)</strong></td>
</tr>
<tr>
<td><strong>posixfile (Unix)</strong></td>
</tr>
<tr>
<td><strong>pprint</strong></td>
</tr>
<tr>
<td><strong>profile</strong></td>
</tr>
<tr>
<td><strong>pstats</strong></td>
</tr>
<tr>
<td><strong>pty (Linux)</strong></td>
</tr>
<tr>
<td><strong>pwd (Unix)</strong></td>
</tr>
<tr>
<td><strong>py_compile</strong></td>
</tr>
<tr>
<td><strong>pyclbr</strong></td>
</tr>
<tr>
<td><strong>pydoc</strong></td>
</tr>
<tr>
<td><strong>q</strong></td>
</tr>
<tr>
<td><strong>quopri</strong></td>
</tr>
<tr>
<td><strong>r</strong></td>
</tr>
<tr>
<td><strong>re</strong></td>
</tr>
<tr>
<td><strong>readline (Unix)</strong></td>
</tr>
<tr>
<td>Module</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>resource (Unix)</td>
</tr>
<tr>
<td>rexec</td>
</tr>
<tr>
<td>rfc822</td>
</tr>
<tr>
<td>rlcompleter</td>
</tr>
<tr>
<td>robotparser</td>
</tr>
<tr>
<td>runpy</td>
</tr>
<tr>
<td>sched</td>
</tr>
<tr>
<td>ScrolledText (Tk)</td>
</tr>
<tr>
<td>select</td>
</tr>
<tr>
<td>sets</td>
</tr>
<tr>
<td>sgmllib</td>
</tr>
<tr>
<td>sha</td>
</tr>
<tr>
<td>shelve</td>
</tr>
<tr>
<td>shlex</td>
</tr>
<tr>
<td>shutil</td>
</tr>
</tbody>
</table>
**signal**

Set handlers for asynchronous events.

**SimpleHTTPServer**

This module provides a basic request handler for HTTP servers.

**SimpleXMLRPCServer**

Basic XML-RPC server implementation.

**site**

Module responsible for site-specific configuration.

**smtpd**

A SMTP server implementation in Python.

**smtplib**

SMTP protocol client (requires sockets).

**sndhdr**

Determine type of a sound file.

**socket**

Low-level networking interface.

**SocketServer**

A framework for network servers.

**spwd (Unix)**

The shadow password database (getspnam() and friends).

**sqlite3**

A DB-API 2.0 implementation using SQLite 3.x.

**ssl**

TLS/SSL wrapper for socket objects

**stat**

Utilities for interpreting the results of os.stat(), os.lstat() and os.fstat().

**statvfs**

**Deprecated:** Constants for interpreting the result of os.statvfs().

**string**

Common string operations.

**StringIO**

Read and write strings as if they were files.

**stringprep**

String preparation, as per RFC 3453
interpret strings as packed binary data.

subprocess

Subprocess management.

sunau

Provide an interface to the Sun AU sound format.

**Deprecated**: Access to Sun audio hardware.

**SUNAUDIODEV** *(SunOS)*

**Deprecated**: Constants for use with sunaudiodev.

symbol

Constants representing internal nodes of the parse tree.

symtable

Interface to the compiler’s internal symbol tables.

sys

Access system-specific parameters and functions.

sysconfig

Python’s configuration information

**syslog** *(Unix)*

An interface to the Unix syslog library routines.

t

**tabnanny**

Tool for detecting white space related problems in Python source files in a directory tree.

**tarfile**

Read and write tar-format archive files.

telnetlib

Telnet client class.

tempfile

Generate temporary files and directories.

termios *(Unix)*

POSIX style tty control.

test *(Unix)*

Regression tests package containing the testing suite for Python.

test.test_support

Support for Python regression
textwrap  
  Text wrapping and filling  

thread  
  Create multiple threads of control within one interpreter.  

threading  
  Higher-level threading interface.  

time  
  Time access and conversions.  

timeit  
  Measure the execution time of small code snippets.  

Tix  
  Tk Extension Widgets for Tkinter  

Tkinter  
  Interface to Tcl/Tk for graphical user interfaces  

token  
  Constants representing terminal nodes of the parse tree.  

tokenize  
  Lexical scanner for Python source code.  

trace  
  Trace or track Python statement execution.  

traceback  
  Print or retrieve a stack traceback.  

ttk  
  Tk themed widget set  

tty (Unix)  
  Utility functions that perform common terminal control operations.  

turtle  
  Turtle graphics for Tk  

types  
  Names for built-in types.  

u  

unicodedata  
  Access the Unicode Database.  

unittest  
  Unit testing framework for Python.  

urllib  
  Open an arbitrary network resource by URL (requires
<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>urllib2</strong></td>
<td>Next generation URL opening library.</td>
</tr>
<tr>
<td><strong>urlparse</strong></td>
<td>Parse URLs into or assemble them from components.</td>
</tr>
<tr>
<td><strong>user</strong></td>
<td>Deprecated: A standard way to reference user-specific modules.</td>
</tr>
<tr>
<td><strong>UserDict</strong></td>
<td>Class wrapper for dictionary objects.</td>
</tr>
<tr>
<td><strong>UserList</strong></td>
<td>Class wrapper for list objects.</td>
</tr>
<tr>
<td><strong>UserString</strong></td>
<td>Class wrapper for string objects.</td>
</tr>
<tr>
<td><strong>uu</strong></td>
<td>Encode and decode files in uuencode format.</td>
</tr>
<tr>
<td><strong>uuid</strong></td>
<td>UUID objects (universally unique identifiers) according to RFC 4122</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>videoreader (Mac)</strong></td>
<td>Deprecated: Read QuickTime movies frame by frame for further processing.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>W</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>w (Mac)</strong></td>
<td>Deprecated: Widgets for the Mac, built on top of FrameWork.</td>
</tr>
<tr>
<td><strong>warnings</strong></td>
<td>Issue warning messages and control their disposition.</td>
</tr>
<tr>
<td><strong>wave</strong></td>
<td>Provide an interface to the WAV sound format.</td>
</tr>
<tr>
<td><strong>weakref</strong></td>
<td>Support for weak references and weak dictionaries.</td>
</tr>
<tr>
<td><strong>webbrowser</strong></td>
<td>Easy-to-use controller for Web browsers.</td>
</tr>
<tr>
<td>Package</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>whichdb</strong></td>
<td>Guess which DBM-style module created a given database.</td>
</tr>
<tr>
<td><strong>winsound (Windows)</strong></td>
<td>Access to the sound-playing machinery for Windows.</td>
</tr>
<tr>
<td><strong>wsgiref</strong></td>
<td>WSGI Utilities and Reference Implementation.</td>
</tr>
<tr>
<td><strong>wsgiref.handlers</strong></td>
<td>WSGI server/gateway base classes.</td>
</tr>
<tr>
<td><strong>wsgiref.headers</strong></td>
<td>WSGI response header tools.</td>
</tr>
<tr>
<td><strong>wsgiref.simple_server</strong></td>
<td>A simple WSGI HTTP server.</td>
</tr>
<tr>
<td><strong>wsgiref.util</strong></td>
<td>WSGI environment utilities.</td>
</tr>
<tr>
<td><strong>wsgiref.validate</strong></td>
<td>WSGI conformance checker.</td>
</tr>
<tr>
<td><strong>x</strong></td>
<td></td>
</tr>
<tr>
<td><strong>xdrplib</strong></td>
<td>Encoders and decoders for the External Data Representation (XDR).</td>
</tr>
<tr>
<td><strong>xml</strong></td>
<td>Package containing XML processing modules</td>
</tr>
<tr>
<td><strong>xml.dom</strong></td>
<td>Document Object Model API for Python.</td>
</tr>
<tr>
<td><strong>xml.dom.minidom</strong></td>
<td>Minimal Document Object Model (DOM) implementation.</td>
</tr>
<tr>
<td><strong>xml.dom.pulldom</strong></td>
<td>Support for building partial DOM trees from SAX events.</td>
</tr>
<tr>
<td><strong>xml.etree.ElementTree</strong></td>
<td>Implementation of the ElementTree API.</td>
</tr>
<tr>
<td><strong>xml.parsers.expat</strong></td>
<td>An interface to the Expat non-validating XML parser.</td>
</tr>
<tr>
<td><strong>xml.sax</strong></td>
<td>Package containing SAX2 base classes and convenience functions.</td>
</tr>
<tr>
<td><strong>xml.sax.handler</strong></td>
<td>Base classes for SAX event handlers.</td>
</tr>
<tr>
<td>Module</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>xml.sax.saxutils</code></td>
<td>Convenience functions and classes for use with SAX.</td>
</tr>
<tr>
<td><code>xml.sax.xmlreader</code></td>
<td>Interface which SAX-compliant XML parsers must implement.</td>
</tr>
<tr>
<td><code>xmlrpcClient</code></td>
<td>XML-RPC client access.</td>
</tr>
<tr>
<td><code>zipfile</code></td>
<td>Read and write ZIP-format archive files.</td>
</tr>
<tr>
<td><code>zipimport</code></td>
<td>Support for importing Python modules from ZIP archives.</td>
</tr>
<tr>
<td><code>zlib</code></td>
<td>Low-level interface to compression and decompression routines compatible with gzip.</td>
</tr>
</tbody>
</table>
This article explains the new features in Python 2.7. Python 2.7 was released on July 3, 2010.

Numeric handling has been improved in many ways, for both floating-point numbers and for the `Decimal` class. There are some useful additions to the standard library, such as a greatly enhanced `unittest` module, the `argparse` module for parsing command-line options, convenient `OrderedDict` and `Counter` classes in the `collections` module, and many other improvements.

Python 2.7 is planned to be the last of the 2.x releases, so we worked on making it a good release for the long term. To help with porting to Python 3, several new features from the Python 3.x series have been included in 2.7.

This article doesn’t attempt to provide a complete specification of the new features, but instead provides a convenient overview. For full details, you should refer to the documentation for Python 2.7 at https://docs.python.org. If you want to understand the rationale for the design and implementation, refer to the PEP for a particular new feature or the issue on https://bugs.python.org in which a change was discussed. Whenever possible, “What’s New in Python” links to the bug/patch item for each change.
The Future for Python 2.x

Python 2.7 is the last major release in the 2.x series, as the Python maintainers have shifted the focus of their new feature development efforts to the Python 3.x series. This means that while Python 2 continues to receive bug fixes, and to be updated to build correctly on new hardware and versions of supported operated systems, there will be no new full feature releases for the language or standard library.

However, while there is a large common subset between Python 2.7 and Python 3, and many of the changes involved in migrating to that common subset, or directly to Python 3, can be safely automated, some other changes (notably those associated with Unicode handling) may require careful consideration, and preferably robust automated regression test suites, to migrate effectively.

This means that Python 2.7 will remain in place for a long time, providing a stable and supported base platform for production systems that have not yet been ported to Python 3. The full expected lifecycle of the Python 2.7 series is detailed in PEP 373.

Some key consequences of the long-term significance of 2.7 are:

- As noted above, the 2.7 release has a much longer period of maintenance when compared to earlier 2.x versions. Python 2.7 is currently expected to remain supported by the core development team (receiving security updates and other bug fixes) until at least 2020 (10 years after its initial release, compared to the more typical support period of 18–24 months).
- As the Python 2.7 standard library ages, making effective use of the Python Package Index (either directly or via a redistributor) becomes more important for Python 2 users. In addition to a wide variety of third party packages for various tasks, the
available packages include backports of new modules and features from the Python 3 standard library that are compatible with Python 2, as well as various tools and libraries that can make it easier to migrate to Python 3. The Python Packaging User Guide provides guidance on downloading and installing software from the Python Package Index.

- While the preferred approach to enhancing Python 2 is now the publication of new packages on the Python Package Index, this approach doesn't necessarily work in all cases, especially those related to network security. In exceptional cases that cannot be handled adequately by publishing new or updated packages on PyPI, the Python Enhancement Proposal process may be used to make the case for adding new features directly to the Python 2 standard library. Any such additions, and the maintenance releases where they were added, will be noted in the New Features Added to Python 2.7 Maintenance Releases section below.

For projects wishing to migrate from Python 2 to Python 3, or for library and framework developers wishing to support users on both Python 2 and Python 3, there are a variety of tools and guides available to help decide on a suitable approach and manage some of the technical details involved. The recommended starting point is the Porting Python 2 Code to Python 3 HOWTO guide.
Changes to the Handling of Deprecation Warnings

For Python 2.7, a policy decision was made to silence warnings only of interest to developers by default. `DeprecationWarning` and its descendants are now ignored unless otherwise requested, preventing users from seeing warnings triggered by an application. This change was also made in the branch that became Python 3.2. (Discussed on stdlib-sig and carried out in issue 7319.)

In previous releases, `DeprecationWarning` messages were enabled by default, providing Python developers with a clear indication of where their code may break in a future major version of Python.

However, there are increasingly many users of Python-based applications who are not directly involved in the development of those applications. `DeprecationWarning` messages are irrelevant to such users, making them worry about an application that's actually working correctly and burdening application developers with responding to these concerns.

You can re-enable display of `DeprecationWarning` messages by running Python with the `-Wdefault` (short form: `-Wd`) switch, or by setting the `PYTHONWARNINGS` environment variable to "default" (or "d") before running Python. Python code can also re-enable them by calling `warnings.simplefilter('default').`

The `unittest` module also automatically reenables deprecation warnings when running tests.
Python 3.1 Features

Much as Python 2.6 incorporated features from Python 3.0, version 2.7 incorporates some of the new features in Python 3.1. The 2.x series continues to provide tools for migrating to the 3.x series.

A partial list of 3.1 features that were backported to 2.7:

- The syntax for set literals (\{1, 2, 3\} is a mutable set).
- Dictionary and set comprehensions (\{i: i*2 for i in range(3)\}).
- Multiple context managers in a single `with` statement.
- A new version of the `io` library, rewritten in C for performance.
- The ordered-dictionary type described in PEP 372: Adding an Ordered Dictionary to collections.
- The new "","" format specifier described in PEP 378: Format Specifier for Thousands Separator.
- The `memoryview` object.
- A small subset of the `importlib` module, described below.
- The `repr()` of a float `x` is shorter in many cases: it’s now based on the shortest decimal string that’s guaranteed to round back to `x`. As in previous versions of Python, it’s guaranteed that `float(repr(x))` recovers `x`.
- Float-to-string and string-to-float conversions are correctly rounded. The `round()` function is also now correctly rounded.
- The `PyCapsule` type, used to provide a C API for extension modules.
- The `PyLong_AsLongAndOverflow()` C API function.

Other new Python3-mode warnings include:

- `operator.isCallable()` and `operator.sequenceIncludes()`,
which are not supported in 3.x, now trigger warnings.

- The `-3` switch now automatically enables the `-Qwarn` switch that causes warnings about using classic division with integers and long integers.
Regular Python dictionaries iterate over key/value pairs in arbitrary order. Over the years, a number of authors have written alternative implementations that remember the order that the keys were originally inserted. Based on the experiences from those implementations, 2.7 introduces a new `OrderedDict` class in the `collections` module.

The `OrderedDict` API provides the same interface as regular dictionaries but iterates over keys and values in a guaranteed order depending on when a key was first inserted:

```
>>> from collections import OrderedDict
>>> d = OrderedDict([('first', 1), ...
...                   ('second', 2), ...
...                   ('third', 3)])
>>> d.items()
[('first', 1), ('second', 2), ('third', 3)]
```

If a new entry overwrites an existing entry, the original insertion position is left unchanged:

```
>>> d['second'] = 4
>>> d.items()
[('first', 1), ('second', 4), ('third', 3)]
```

Deleting an entry and reinserting it will move it to the end:

```
>>> del d['second']
>>> d['second'] = 5
>>> d.items()
[('first', 1), ('third', 3), ('second', 5)]
```

The `popitem()` method has an optional `last` argument that defaults to
True. If last is true, the most recently added key is returned and removed; if it's false, the oldest key is selected:

```python
>>> od = OrderedDict([(x, 0) for x in range(20)])
>>> od.popitem()
(19, 0)
>>> od.popitem()
(18, 0)
>>> od.popitem(last=False)
(0, 0)
>>> od.popitem(last=False)
(1, 0)
```

Comparing two ordered dictionaries checks both the keys and values, and requires that the insertion order was the same:

```python
>>> od1 = OrderedDict([('first', 1),
...                      ('second', 2),
...                      ('third', 3)])
>>> od2 = OrderedDict([('third', 3),
...                      ('first', 1),
...                      ('second', 2)])
>>> od1 == od2
False
>>> # Move 'third' key to the end
>>> del od2['third']; od2['third'] = 3
>>> od1 == od2
True
```

Comparing an `OrderedDict` with a regular dictionary ignores the insertion order and just compares the keys and values.

How does the `OrderedDict` work? It maintains a doubly-linked list of keys, appending new keys to the list as they’re inserted. A secondary dictionary maps keys to their corresponding list node, so deletion doesn’t have to traverse the entire linked list and therefore remains O(1).

The standard library now supports use of ordered dictionaries in several modules.
- The `ConfigParser` module uses them by default, meaning that configuration files can now be read, modified, and then written back in their original order.
- The `_asdict()` method for `collections.namedtuple()` now returns an ordered dictionary with the values appearing in the same order as the underlying tuple indices.
- The `json` module’s `JSONDecoder` class constructor was extended with an `object_pairs_hook` parameter to allow `OrderedDict` instances to be built by the decoder. Support was also added for third-party tools like PyYAML.

**See also:**

**PEP 372 - Adding an ordered dictionary to collections**

PEP written by Armin Ronacher and Raymond Hettinger; implemented by Raymond Hettinger.
PEP 378: Format Specifier for Thousands Separator

To make program output more readable, it can be useful to add separators to large numbers, rendering them as 18,446,744,073,709,551,616 instead of 18446744073709551616.

The fully general solution for doing this is the `locale` module, which can use different separators ("," in North America, "." in Europe) and different grouping sizes, but `locale` is complicated to use and unsuitable for multi-threaded applications where different threads are producing output for different locales.

Therefore, a simple comma-grouping mechanism has been added to the mini-language used by the `str.format()` method. When formatting a floating-point number, simply include a comma between the width and the precision:

```python
>>> '{:20,.2f}'.format(18446744073709551616.0)
'18,446,744,073,709,551,616.00'
```

When formatting an integer, include the comma after the width:

```python
>>> '{:20,d}'.format(18446744073709551616)
'18,446,744,073,709,551,616'
```

This mechanism is not adaptable at all; commas are always used as the separator and the grouping is always into three-digit groups. The comma-formatting mechanism isn't as general as the `locale` module, but it's easier to use.

See also:

PEP 378 - Format Specifier for Thousands Separator
PEP written by Raymond Hettinger; implemented by Eric Smith.
The `argparse` module for parsing command-line arguments was added as a more powerful replacement for the `optparse` module.

This means Python now supports three different modules for parsing command-line arguments: `getopt`, `optparse`, and `argparse`. The `getopt` module closely resembles the C library's `getopt()` function, so it remains useful if you're writing a Python prototype that will eventually be rewritten in C. `optparse` becomes redundant, but there are no plans to remove it because there are many scripts still using it, and there's no automated way to update these scripts. (Making the `argparse` API consistent with `optparse`'s interface was discussed but rejected as too messy and difficult.)

In short, if you’re writing a new script and don’t need to worry about compatibility with earlier versions of Python, use `argparse` instead of `optparse`.

Here’s an example:

```python
import argparse

parser = argparse.ArgumentParser(description='Command-line example.'

# Add optional switches
parser.add_argument('-v', action='store_true', dest='is_verbose',
                    help='produce verbose output')
parser.add_argument('-o', action='store', dest='output',
                    metavar='FILE',
                    help='direct output to FILE instead of stdout')
parser.add_argument('-C', action='store', type=int, dest='context',
                    metavar='NUM', default=0,
                    help='display NUM lines of added context')
```
# Allow any number of additional arguments.
parser.add_argument(nargs='*', action='store', dest='inputs',
    help='input filenames (default is stdin)')

args = parser.parse_args()
print(args.__dict__)

Unless you override it, -h and --help switches are automatically added, and produce neatly formatted output:

```
> ./python.exe argparse-example.py --help
usage: argparse-example.py [-h] [-v] [-o FILE] [-C NUM] [inputs
Command-line example.

positional arguments:
  inputs   input filenames (default is stdin)

optional arguments:
  -h, --help     show this help message and exit
  -v             produce verbose output
  -o FILE        direct output to FILE instead of stdout
  -C NUM         display NUM lines of added context
```

As with `optparse`, the command-line switches and arguments are returned as an object with attributes named by the `dest` parameters:

```
> ./python.exe argparse-example.py -v
{'output': None,
 'is_verbos': True,
 'context': 0,
 'inputs': []}

> ./python.exe argparse-example.py -v -o /tmp/output -C 4 file
{'output': '/tmp/output',
 'is_verbos': True,
 'context': 4,
 'inputs': ['file1', 'file2']}
```

`argparse` has much fancier validation than `optparse`; you can specify
an exact number of arguments as an integer, 0 or more arguments by passing '*', 1 or more by passing '+', or an optional argument with '?'. A top-level parser can contain sub-parsers to define subcommands that have different sets of switches, as in `svn commit`, `svn checkout`, etc. You can specify an argument's type as `FileType`, which will automatically open files for you and understands that '-' means standard input or output.

See also:

**argparse documentation**
The documentation page of the argparse module.

**Upgrading optparse code**
Part of the Python documentation, describing how to convert code that uses `optparse`.

**PEP 389 - argparse - New Command Line Parsing Module**
PEP written and implemented by Steven Bethard.
PEP 391: Dictionary-Based Configuration For Logging

The `logging` module is very flexible; applications can define a tree of logging subsystems, and each logger in this tree can filter out certain messages, format them differently, and direct messages to a varying number of handlers.

All this flexibility can require a lot of configuration. You can write Python statements to create objects and set their properties, but a complex set-up requires verbose but boring code. `logging` also supports a `fileConfig()` function that parses a file, but the file format doesn’t support configuring filters, and it’s messier to generate programmatically.

Python 2.7 adds a `dictConfig()` function that uses a dictionary to configure logging. There are many ways to produce a dictionary from different sources: construct one with code; parse a file containing JSON; or use a YAML parsing library if one is installed. For more information see Configuration functions.

The following example configures two loggers, the root logger and a logger named “network”. Messages sent to the root logger will be sent to the system log using the syslog protocol, and messages to the “network” logger will be written to a `network.log` file that will be rotated once the log reaches 1MB.

```python
import logging
import logging.config

configdict = {
    'version': 1,  # Configuration schema in use; must be 1 for
    'formatters': {
        'standard': {
            'format': ('%(asctime)s %name)-15s '        
```
Three smaller enhancements to the `logging` module, all implemented by Vinay Sajip, are:

- The `SysLogHandler` class now supports syslogging over TCP. The constructor has a `socktype` parameter giving the type of socket to use, either `socket.SOCK_DGRAM` for UDP or `socket.SOCK_STREAM` for TCP. The default protocol remains UDP.
- `Logger` instances gained a `getChild()` method that retrieves a
descendant logger using a relative path. For example, once you retrieve a logger by doing `log = getLogger('app')`, calling `log.getChild('network.listen')` is equivalent to `getLogger('app.network.listen')`.

- The LoggerAdapter class gained an `isEnabledFor()` method that takes a `level` and returns whether the underlying logger would process a message of that level of importance.

See also:

**PEP 391 - Dictionary-Based Configuration For Logging**

PEP written and implemented by Vinay Sajip.
PEP 3106: Dictionary Views

The dictionary methods `keys()`, `values()`, and `items()` are different in Python 3.x. They return an object called a *view* instead of a fully materialized list.

It's not possible to change the return values of `keys()`, `values()`, and `items()` in Python 2.7 because too much code would break. Instead the 3.x versions were added under the new names `viewkeys()`, `viewvalues()`, and `viewitems()`.

```python
>>> d = dict((i*10, chr(65+i)) for i in range(26))
>>> d
>>> d.viewkeys()
dict_keys([0, 130, 10, 140, 20, 150, 30, ..., 250])
```

Views can be iterated over, but the key and item views also behave like sets. The `&` operator performs intersection, and `|` performs a union:

```python
>>> d1 = dict((i*10, chr(65+i)) for i in range(26))
>>> d2 = dict((i**.5, i) for i in range(1000))
>>> d1.viewkeys() & d2.viewkeys()
set([0.0, 10.0, 20.0, 30.0])
>>> d1.viewkeys() | range(0, 30)
set([0, 1, 130, 3, 4, 5, 6, ..., 120, 250])
```

The view keeps track of the dictionary and its contents change as the dictionary is modified:

```python
>>> vk = d.viewkeys()
>>> vk
dict_keys([0, 130, 10, ..., 250])
>>> d[260] = '&'
>>> vk
dict_keys([0, 130, 260, 10, ..., 250])
```
However, note that you can’t add or remove keys while you’re iterating over the view:

```python
>>> for k in vk:
...     d[k*2] = k
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```

You can use the view methods in Python 2.x code, and the 2to3 converter will change them to the standard `keys()`, `values()`, and `items()` methods.

See also:

**PEP 3106 - Revamping `dict.keys()`, `values()` and `items()`**

PEP written by Guido van Rossum. Backported to 2.7 by Alexandre Vassalotti; issue 1967.
The `memoryview` object provides a view of another object’s memory content that matches the `bytes` type’s interface.

```python
>>> import string
>>> m = memoryview(string.letters)
>>> m
<memory at 0x37f850>
>>> len(m)  # Returns length of underlying object
52
>>> m[0], m[25], m[26]  # Indexing returns one byte
('a', 'z', 'A')
>>> m2 = m[0:26]  # Slicing returns another memoryview
>>> m2
<memory at 0x37f080>
```

The content of the view can be converted to a string of bytes or a list of integers:

```python
>>> m2.tobytes()
'abcdefghijklmnopqrstuvwxyz'
>>> m2.tolist()
[97, 98, 99, 100, 101, 102, 103, ..., 121, 122]
```

`memoryview` objects allow modifying the underlying object if it’s a mutable object.

```python
>>> m2[0] = 75
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
>>> b = bytearray(string.letters)  # Creating a mutable object
>>> b
bytearray(b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ')
>>> mb = memoryview(b)
>>> mb[0] = '*'  # Assign to view, changing the bytearray
>>> b[0:5]  # The bytearray has been changed.
bytearray(b'*bcde')
```
See also:

**PEP 3137 - Immutable Bytes and Mutable Buffer**
PEP written by Guido van Rossum. Implemented by Travis Oliphant, Antoine Pitrou and others. Backported to 2.7 by Antoine Pitrou; issue 2396.
Other Language Changes

Some smaller changes made to the core Python language are:

- The syntax for set literals has been backported from Python 3.x. Curly brackets are used to surround the contents of the resulting mutable set; set literals are distinguished from dictionaries by not containing colons and values. `{}` continues to represent an empty dictionary; use `set()` for an empty set.

```
>>> {1, 2, 3, 4, 5}
set([1, 2, 3, 4, 5])
>>> set() # empty set
set([])
>>> {} # empty dict
{}
```

Backported by Alexandre Vassalotti; issue 2335.

- Dictionary and set comprehensions are another feature backported from 3.x, generalizing list/generator comprehensions to use the literal syntax for sets and dictionaries.

```
>>> {x: x*x for x in range(6)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
>>> {'a'*x for x in range(6)}
set([', 'a', 'aa', 'aaa', 'aaaa', 'aaaaa'])
```

Backported by Alexandre Vassalotti; issue 2333.

- The `with` statement can now use multiple context managers in one statement. Context managers are processed from left to right and each one is treated as beginning a new `with` statement. This means that:

```
with A() as a, B() as b:
```
... suite of statements ...

is equivalent to:

```python
with A() as a:
    with B() as b:
        ... suite of statements ...
```

The `contextlib.nested()` function provides a very similar function, so it’s no longer necessary and has been deprecated.

(Proposed in https://codereview.appspot.com/53094; implemented by Georg Brandl.)

- Conversions between floating-point numbers and strings are now correctly rounded on most platforms. These conversions occur in many different places: `str()` on floats and complex numbers; the `float` and `complex` constructors; numeric formatting; serializing and deserializing floats and complex numbers using the `marshal`, `pickle` and `json` modules; parsing of float and imaginary literals in Python code; and `Decimal`-to-float conversion.

Related to this, the `repr()` of a floating-point number `x` now returns a result based on the shortest decimal string that's guaranteed to round back to `x` under correct rounding (with round-half-to-even rounding mode). Previously it gave a string based on rounding `x` to 17 decimal digits.

The rounding library responsible for this improvement works on Windows and on Unix platforms using the gcc, icc, or suncc compilers. There may be a small number of platforms where correct operation of this code cannot be guaranteed, so the code is not used on such systems. You can find out which code is being used by checking `sys.float_repr_style`, which will be
short if the new code is in use and legacy if it isn’t.

Implemented by Eric Smith and Mark Dickinson, using David Gay’s dtoa.c library; issue 7117.

- Conversions from long integers and regular integers to floating point now round differently, returning the floating-point number closest to the number. This doesn’t matter for small integers that can be converted exactly, but for large numbers that will unavoidably lose precision, Python 2.7 now approximates more closely. For example, Python 2.6 computed the following:

```python
>>> n = 295147905179352891391
>>> float(n)
2.9514790517935283e+20
>>> n - long(float(n))
65535L
```

Python 2.7’s floating-point result is larger, but much closer to the true value:

```python
>>> n = 295147905179352891391
>>> float(n)
2.9514790517935289e+20
>>> n - long(float(n))
-1L
```

(Implemented by Mark Dickinson; issue 3166.)

- Integer division is also more accurate in its rounding behaviours. (Also implemented by Mark Dickinson; issue 1811.)

- Implicit coercion for complex numbers has been removed; the interpreter will no longer ever attempt to call a `__coerce__()` method on complex objects. (Removed by Meador Inge and Mark Dickinson; issue 5211.)

- The `str.format()` method now supports automatic numbering of
the replacement fields. This makes using `str.format()` more closely resemble using `%s` formatting:

```python
>>> '{0}:{1}:{2}'.format(2009, 04, 'Sunday')
'2009:4:Sunday'
>>> '{0}:{1}:{day}'.format(2009, 4, day='Sunday')
'2009:4:Sunday'
```

The auto-numbering takes the fields from left to right, so the first `{...}` specifier will use the first argument to `str.format()`, the next specifier will use the next argument, and so on. You can’t mix auto-numbering and explicit numbering – either number all of your specifier fields or none of them – but you can mix auto-numbering and named fields, as in the second example above. (Contributed by Eric Smith; issue 5237.)

Complex numbers now correctly support usage with `format()`, and default to being right-aligned. Specifying a precision or comma-separation applies to both the real and imaginary parts of the number, but a specified field width and alignment is applied to the whole of the resulting `1.5+3j` output. (Contributed by Eric Smith; issue 1588 and issue 7988.)

The ‘F’ format code now always formats its output using uppercase characters, so it will now produce ‘INF’ and ‘NAN’. (Contributed by Eric Smith; issue 3382.)

A low-level change: the `object.__format__() method now triggers a `PendingDeprecationWarning` if it’s passed a format string, because the `__format__()` method for `object` converts the object to a string representation and formats that. Previously the method silently applied the format string to the string representation, but that could hide mistakes in Python code. If you’re supplying formatting information such as an alignment or precision, presumably you’re expecting the formatting to be
applied in some object-specific way. (Fixed by Eric Smith; issue 7994.)

- The `int()` and `long()` types gained a `bit_length` method that returns the number of bits necessary to represent its argument in binary:

  ```python
  >>> n = 37
  >>> bin(n)
  '0b100101'
  >>> n.bit_length()
  6
  >>> n = 2**123-1
  >>> n.bit_length()
  123
  >>> (n+1).bit_length()
  124
  ```

  (Contributed by Fredrik Johansson and Victor Stinner; issue 3439.)

- The `import` statement will no longer try an absolute import if a relative import (e.g. `from .os import sep`) fails. This fixes a bug, but could possibly break certain `import` statements that were only working by accident. (Fixed by Meador Inge; issue 7902.)

- It's now possible for a subclass of the built-in `unicode` type to override the `__unicode__()` method. (Implemented by Victor Stinner; issue 1583863.)

- The `bytearray` type's `translate()` method now accepts `None` as its first argument. (Fixed by Georg Brandl; issue 4759.)

- When using `@classmethod` and `@staticmethod` to wrap methods as class or static methods, the wrapper object now exposes the wrapped function as their `__func__` attribute. (Contributed by Amaury Forgeot d'Arc, after a suggestion by George Sakkis;
When a restricted set of attributes were set using __slots__, deleting an unset attribute would not raise AttributeError as you would expect. Fixed by Benjamin Peterson; issue 7604.)

Two new encodings are now supported: “cp720”, used primarily for Arabic text; and “cp858”, a variant of CP 850 that adds the euro symbol. (CP720 contributed by Alexander Belchenko and Amaury Forgeot d'Arc in issue 1616979; CP858 contributed by Tim Hatch in issue 8016.)

The file object will now set the filename attribute on the IOError exception when trying to open a directory on POSIX platforms (noted by Jan Kaliszewski; issue 4764), and now explicitly checks for and forbids writing to read-only file objects instead of trusting the C library to catch and report the error (fixed by Stefan Krah; issue 5677).

The Python tokenizer now translates line endings itself, so the compile() built-in function now accepts code using any line-ending convention. Additionally, it no longer requires that the code end in a newline.

Extra parentheses in function definitions are illegal in Python 3.x, meaning that you get a syntax error from def f((x)): pass. In Python3-warning mode, Python 2.7 will now warn about this odd usage. (Noted by James Lingard; issue 7362.)

It’s now possible to create weak references to old-style class objects. New-style classes were always weak-referenceable. (Fixed by Antoine Pitrou; issue 8268.)

When a module object is garbage-collected, the module’s dictionary is now only cleared if no one else is holding a
reference to the dictionary (issue 7140).

## Interpreter Changes

A new environment variable, `PYTHONWARNINGS`, allows controlling warnings. It should be set to a string containing warning settings, equivalent to those used with the `-w` switch, separated by commas. (Contributed by Brian Curtin; issue 7301.)

For example, the following setting will print warnings every time they occur, but turn warnings from the `Cookie` module into an error. (The exact syntax for setting an environment variable varies across operating systems and shells.)

```bash
export PYTHONWARNINGS=all,error::Cookie:0
```

## Optimizations

Several performance enhancements have been added:

- A new opcode was added to perform the initial setup for `with` statements, looking up the `__enter__()` and `__exit__()` methods. (Contributed by Benjamin Peterson.)

- The garbage collector now performs better for one common usage pattern: when many objects are being allocated without deallocating any of them. This would previously take quadratic time for garbage collection, but now the number of full garbage collections is reduced as the number of objects on the heap grows. The new logic only performs a full garbage collection pass when the middle generation has been collected 10 times and when the number of survivor objects from the middle generation exceeds 10% of the number of objects in the oldest generation. (Suggested by Martin von Löwis and implemented
by Antoine Pitrou; issue 4074.)

- The garbage collector tries to avoid tracking simple containers which can’t be part of a cycle. In Python 2.7, this is now true for tuples and dicts containing atomic types (such as ints, strings, etc.). Transitively, a dict containing tuples of atomic types won’t be tracked either. This helps reduce the cost of each garbage collection by decreasing the number of objects to be considered and traversed by the collector. (Contributed by Antoine Pitrou; issue 4688.)

- Long integers are now stored internally either in base $2^{15}$ or in base $2^{30}$, the base being determined at build time. Previously, they were always stored in base $2^{15}$. Using base $2^{30}$ gives significant performance improvements on 64-bit machines, but benchmark results on 32-bit machines have been mixed. Therefore, the default is to use base $2^{30}$ on 64-bit machines and base $2^{15}$ on 32-bit machines; on Unix, there’s a new configure option --enable-big-digits that can be used to override this default.

Apart from the performance improvements this change should be invisible to end users, with one exception: for testing and debugging purposes there’s a new structseq sys.long_info that provides information about the internal format, giving the number of bits per digit and the size in bytes of the C type used to store each digit:

```python
>>> import sys
>>> sys.long_info
sys.long_info(bits_per_digit=30, sizeof_digit=4)
```

(Contributed by Mark Dickinson; issue 4258.)

Another set of changes made long objects a few bytes smaller: 2 bytes smaller on 32-bit systems and 6 bytes on 64-bit.
The division algorithm for long integers has been made faster by tightening the inner loop, doing shifts instead of multiplications, and fixing an unnecessary extra iteration. Various benchmarks show speedups of between 50% and 150% for long integer divisions and modulo operations. (Contributed by Mark Dickinson; issue 5512.) Bitwise operations are also significantly faster (initial patch by Gregory Smith; issue 1087418).

The implementation of % checks for the left-side operand being a Python string and special-cases it; this results in a 1–3% performance increase for applications that frequently use % with strings, such as templating libraries. (Implemented by Collin Winter; issue 5176.)

List comprehensions with an if condition are compiled into faster bytecode. (Patch by Antoine Pitrou, back-ported to 2.7 by Jeffrey Yasskin; issue 4715.)

Converting an integer or long integer to a decimal string was made faster by special-casing base 10 instead of using a generalized conversion function that supports arbitrary bases. (Patch by Gawain Bolton; issue 6713.)

The split(), replace(), rindex(), rpartition(), and rsplit() methods of string-like types (strings, Unicode strings, and bytearray objects) now use a fast reverse-search algorithm instead of a character-by-character scan. This is sometimes faster by a factor of 10. (Added by Florent Xicluna; issue 7462 and issue 7622.)

The pickle and cPickle modules now automatically intern the strings used for attribute names, reducing memory usage of the objects resulting from unpickling. (Contributed by Jake McGuire;
issue 5084.)

- The **cPickle** module now special-cases dictionaries, nearly halving the time required to pickle them. (Contributed by Collin Winter; issue 5670.)
New and Improved Modules

As in every release, Python’s standard library received a number of enhancements and bug fixes. Here’s a partial list of the most notable changes, sorted alphabetically by module name. Consult the Misc/NEWS file in the source tree for a more complete list of changes, or look through the Subversion logs for all the details.

- The bdb module’s base debugging class Bdb gained a feature for skipping modules. The constructor now takes an iterable containing glob-style patterns such as django.*; the debugger will not step into stack frames from a module that matches one of these patterns. (Contributed by Maru Newby after a suggestion by Senthil Kumaran; issue 5142.)

- The binascii module now supports the buffer API, so it can be used with memoryview instances and other similar buffer objects. (Backported from 3.x by Florent Xicluna; issue 7703.)

- Updated module: the bsddb module has been updated from 4.7.2devel9 to version 4.8.4 of the pybsddb package. The new version features better Python 3.x compatibility, various bug fixes, and adds several new BerkeleyDB flags and methods. (Updated by Jesús Cea Avión; issue 8156. The pybsddb changelog can be read at http://hg.jcea.es/pybsddb/file/tip/ChangeLog.)

- The bz2 module’s BZ2File now supports the context management protocol, so you can write with bz2.BZ2File(...) as f:. (Contributed by Hagen Fürstenau; issue 3860.)

- New class: the Counter class in the collections module is useful for tallying data. Counter instances behave mostly like
dictionaries but return zero for missing keys instead of raising a
KeyError:

```
>>> from collections import Counter
>>> c = Counter()
>>> for letter in 'here is a sample of english text':
...     c[letter] += 1
...     
>>> c
Counter({' ': 6, 'e': 5, 's': 3, 'a': 2, 'i': 2, 'h': 2, 'l': 2, 't': 2, 'g': 1, 'f': 1, 'm': 1, 'o': 1, 'n': 1, 'p': 1, 'r': 1, 'x': 1})
>>> c['e']
5
>>> c['z']
0
```

There are three additional `Counter` methods. `most_common()`
returns the N most common elements and their counts. `elements()`
returns an iterator over the contained elements, repeating each
element as many times as its count. `subtract()`
takes an iterable and subtracts one for each element instead of
adding; if the argument is a dictionary or another `Counter`, the
counts are subtracted.

```
>>> c.most_common(5)
[(' ', 6), ('e', 5), ('s', 3), ('a', 2), ('i', 2)]
>>> c.elements() ->
'a', 'a', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
'e', 'e', 'e', 'e', 'e', 'g', 'f', 'i', 'i', 'i',
'h', 'h', 'm', 'l', 'l', 'o', 'n', 'p', 's',
's', 's', 'r', 't', 't', 'x'
>>> c['e']
5
>>> c.subtract('very heavy on the letter e')
>>> c['e']    # Count is now lower
-1
```

Contributed by Raymond Hettinger; issue 1696199.
New class: *OrderedDict* is described in the earlier section PEP 372: Adding an Ordered Dictionary to collections.

New method: The *deque* data type now has a *count()* method that returns the number of contained elements equal to the supplied argument \( x \), and a *reverse()* method that reverses the elements of the deque in-place. *deque* also exposes its maximum length as the read-only *maxlen* attribute. (Both features added by Raymond Hettinger.)

The *namedtuple* class now has an optional *rename* parameter. If *rename* is true, field names that are invalid because they've been repeated or aren't legal Python identifiers will be renamed to legal names that are derived from the field's position within the list of fields:

```python
>>> from collections import namedtuple
>>> T = namedtuple('T', ['field1', '$illegal', 'for', 'field2']
>>> T._fields
('field1', '_1', '_2', 'field2')
```

(Added by Raymond Hettinger; issue 1818.)

Finally, the *Mapping* abstract base class now returns *NotImplemented* if a mapping is compared to another type that isn't a *Mapping*. (Fixed by Daniel Stutzbach; issue 8729.)

- Constructors for the parsing classes in the *ConfigParser* module now take an *allow_no_value* parameter, defaulting to false; if true, options without values will be allowed. For example:

```python
>>> import ConfigParser, StringIO
>>> sample_config = ""
... [mysql]
... user = mysql
... pid-file = /var/run/mysqld/mysqld.pid
```
... skip-bdb
...

```python
>>> config = ConfigParser.RawConfigParser(allow_no_value=True)
>>> config.readfp(StringIO.StringIO(sample_config))
>>> config.get('mysqld', 'user')
'mysql'
>>> print config.get('mysqld', 'skip-bdb')
None
>>> print config.get('mysqld', 'unknown')
Traceback (most recent call last):
...
NoOptionError: No option 'unknown' in section: 'mysqld'
```

(Contributed by Mats Kindahl; issue 7005.)

- Deprecated function: `contextlib.nested()`, which allows handling more than one context manager with a single `with` statement, has been deprecated, because the `with` statement now supports multiple context managers.

- The `cookielib` module now ignores cookies that have an invalid version field, one that doesn’t contain an integer value. (Fixed by John J. Lee; issue 3924.)

- The `copy` module's `deepcopy()` function will now correctly copy bound instance methods. (Implemented by Robert Collins; issue 1515.)

- The `ctypes` module now always converts `None` to a C NULL pointer for arguments declared as pointers. (Changed by Thomas Heller; issue 4606.) The underlying `libffi` library has been updated to version 3.0.9, containing various fixes for different platforms. (Updated by Matthias Klose; issue 8142.)

- New method: the `datetime` module's `timedelta` class gained a `total_seconds()` method that returns the number of seconds in the duration. (Contributed by Brian Quinlan; issue 5788.)
New method: the **Decimal** class gained a **from_float()** class method that performs an exact conversion of a floating-point number to a **Decimal**. This exact conversion strives for the closest decimal approximation to the floating-point representation’s value; the resulting decimal value will therefore still include the inaccuracy, if any. For example, `Decimal.from_float(0.1)` returns `Decimal('0.10000000000000000555111512312578270211815834045410')` (Implemented by Raymond Hettinger; [issue 4796](https://bugs.python.org/issue4796).)

Comparing instances of **Decimal** with floating-point numbers now produces sensible results based on the numeric values of the operands. Previously such comparisons would fall back to Python’s default rules for comparing objects, which produced arbitrary results based on their type. Note that you still cannot combine **Decimal** and floating-point in other operations such as addition, since you should be explicitly choosing how to convert between float and **Decimal**. (Fixed by Mark Dickinson; [issue 2531](https://bugs.python.org/issue2531).)

The constructor for **Decimal** now accepts floating-point numbers (added by Raymond Hettinger; [issue 8257](https://bugs.python.org/issue8257)) and non-European Unicode characters such as Arabic-Indic digits (contributed by Mark Dickinson; [issue 6595](https://bugs.python.org/issue6595)).

Most of the methods of the **Context** class now accept integers as well as **Decimal** instances; the only exceptions are the **canonical()** and **is_canonical()** methods. (Patch by Juan José Conti; [issue 7633](https://bugs.python.org/issue7633).)

When using **Decimal** instances with a string’s **format()** method, the default alignment was previously left-alignment. This has been changed to right-alignment, which is more sensible for numeric types. (Changed by Mark Dickinson; [issue 6857](https://bugs.python.org/issue6857).)
Comparisons involving a signaling NaN value (or \texttt{SNAN}) now signal \texttt{InvalidOperation} instead of silently returning a true or false value depending on the comparison operator. Quiet NaN values (or \texttt{NaN}) are now hashable. (Fixed by Mark Dickinson; \texttt{issue 7279}.)

- The \texttt{difflib} module now produces output that is more compatible with modern \texttt{diff/patch} tools through one small change, using a tab character instead of spaces as a separator in the header giving the filename. (Fixed by Anatoly Techtonik; \texttt{issue 7585}.)

- The \texttt{distutils sdist} command now always regenerates the \texttt{MANIFEST} file, since even if the \texttt{MANIFEST.in} or \texttt{setup.py} files haven’t been modified, the user might have created some new files that should be included. (Fixed by Tarek Ziadé; \texttt{issue 8688}.)

- The \texttt{doctest} module’s \texttt{IGNORE_EXCEPTION_DETAIL} flag will now ignore the name of the module containing the exception being tested. (Patch by Lennart Regebro; \texttt{issue 7490}.)

- The \texttt{email} module’s \texttt{Message} class will now accept a Unicode-valued payload, automatically converting the payload to the encoding specified by \texttt{output_charset}. (Added by R. David Murray; \texttt{issue 1368247}.)

- The \texttt{Fraction} class now accepts a single float or \texttt{Decimal} instance, or two rational numbers, as arguments to its constructor. (Implemented by Mark Dickinson; rationals added in \texttt{issue 5812}, and float/decimal in \texttt{issue 8294}.)

Ordering comparisons (\texttt{<, \leq, >, \geq}) between fractions and complex numbers now raise a \texttt{TypeError}. This fixes an oversight, making the \texttt{Fraction} match the other numeric types.
• New class: `FTP_TLS` in the `ftplib` module provides secure FTP connections using TLS encapsulation of authentication as well as subsequent control and data transfers. (Contributed by Giampaolo Rodola; issue 2054.)

The `storbinary()` method for binary uploads can now restart uploads thanks to an added `rest` parameter (patch by Pablo Mouzo; issue 6845.)

• New class decorator: `total_ordering()` in the `functools` module takes a class that defines an `__eq__()` method and one of `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`, and generates the missing comparison methods. Since the `__cmp__()` method is being deprecated in Python 3.x, this decorator makes it easier to define ordered classes. (Added by Raymond Hettinger; issue 5479.)

New function: `cmp_to_key()` will take an old-style comparison function that expects two arguments and return a new callable that can be used as the `key` parameter to functions such as `sorted()`, `min()` and `max()`, etc. The primary intended use is to help with making code compatible with Python 3.x. (Added by Raymond Hettinger.)

• New function: the `gc` module’s `is_tracked()` returns true if a given instance is tracked by the garbage collector, false otherwise. (Contributed by Antoine Pitrou; issue 4688.)

• The `gzip` module’s `GzipFile` now supports the context management protocol, so you can write `with gzip.GzipFile(...) as f:` (contributed by Hagen Fürstenau; issue 3860), and it now implements the `io.BufferedIOBase` ABC, so you can wrap it with `io.BufferedReader` for faster processing (contributed by Nir Aides; issue 7471). It's also now possible to
override the modification time recorded in a gzipped file by providing an optional timestamp to the constructor. (Contributed by Jacques Frechet; issue 4272.)

Files in gzip format can be padded with trailing zero bytes; the gzip module will now consume these trailing bytes. (Fixed by Tadek Pietraszek and Brian Curtin; issue 2846.)

- New attribute: the hashlib module now has an algorithms attribute containing a tuple naming the supported algorithms. In Python 2.7, hashlib.algorithms contains ('md5', 'sha1', 'sha224', 'sha256', 'sha384', 'sha512'). (Contributed by Carl Chenet; issue 7418.)

- The default HTTPResponse class used by the httplib module now supports buffering, resulting in much faster reading of HTTP responses. (Contributed by Kristján Valur Jónsson; issue 4879.)

The HTTPConnection and HTTPSConnection classes now support a source_address parameter, a (host, port) 2-tuple giving the source address that will be used for the connection. (Contributed by Eldon Ziegler; issue 397.)

- The ihooks module now supports relative imports. Note that ihooks is an older module for customizing imports, superseded by the imputil module added in Python 2.0. (Relative import support added by Neil Schemenauer.)

- The imaplib module now supports IPv6 addresses. (Contributed by Derek Morr; issue 1655.)

- New function: the inspect module’s getcallargs() takes a callable and its positional and keyword arguments, and figures
out which of the callable's parameters will receive each argument, returning a dictionary mapping argument names to their values. For example:

```python
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
>>> getcallargs(f, 1, 2, 3)
{'a': 1, 'b': 2, 'pos': (3,), 'named': {}}
>>> getcallargs(f, a=2, x=4)
{'a': 2, 'b': 1, 'pos': (), 'named': {'x': 4}}
>>> getcallargs(f)
Traceback (most recent call last):
...  TypeError: f() takes at least 1 argument (0 given)
```

Contributed by George Sakkis; issue 3135.

- Updated module: The `io` library has been upgraded to the version shipped with Python 3.1. For 3.1, the I/O library was entirely rewritten in C and is 2 to 20 times faster depending on the task being performed. The original Python version was renamed to the `_pyio` module.

One minor resulting change: the `io.TextIOWrapper` class now has an `errors` attribute giving the error setting used for encoding and decoding errors (one of 'strict', 'replace', 'ignore').

The `io.FileIO` class now raises an `OSError` when passed an invalid file descriptor. (Implemented by Benjamin Peterson; issue 4991.) The `truncate()` method now preserves the file position; previously it would change the file position to the end of the new file. (Fixed by Pascal Chambon; issue 6939.)

- New function: `itertools.compress(data, selectors) takes two iterators. Elements of `data` are returned if the corresponding value in `selectors` is true:
New function: `itertools.combinations_with_replacement(iter, r)` returns all the possible `r`-length combinations of elements from the iterable `iter`. Unlike `combinations()`, individual elements can be repeated in the generated combinations:

```
>>> itertools.combinations_with_replacement('ABCDEF', [1,0,1,0,1,1])
A, C, E, F
```

Note that elements are treated as unique depending on their position in the input, not their actual values.

The `itertools.count()` function now has a `step` argument that allows incrementing by values other than 1. `count()` also now allows keyword arguments, and using non-integer values such as floats or `Decimal` instances. (Implemented by Raymond Hettinger; issue 5032.)

`itertools.combinations()` and `itertools.product()` previously raised `ValueError` for values of `r` larger than the input iterable. This was deemed a specification error, so they now return an empty iterator. (Fixed by Raymond Hettinger; issue 4816.)

- Updated module: The `json` module was upgraded to version 2.0.9 of the simplejson package, which includes a C extension that makes encoding and decoding faster. (Contributed by Bob Ippolito; issue 4136.)

To support the new `collections.OrderedDict` type, `json.load()` now has an optional `object_pairs_hook` parameter that will be called with any object literal that decodes to a list of pairs. (Contributed by Raymond Hettinger; issue 5381.)
- The `mailbox` module's `Maildir` class now records the timestamp on the directories it reads, and only re-reads them if the modification time has subsequently changed. This improves performance by avoiding unneeded directory scans. (Fixed by A.M. Kuchling and Antoine Pitrou; issue 1607951, issue 6896.)

- New functions: the `math` module gained `erf()` and `erfc()` for the error function and the complementary error function, `expm1()` which computes $e^{x} - 1$ with more precision than using `exp()` and subtracting 1, `gamma()` for the Gamma function, and `lgamma()` for the natural log of the Gamma function. (Contributed by Mark Dickinson and nirinA raseliarison; issue 3366.)

- The `multiprocessing` module's `Manager*` classes can now be passed a callable that will be called whenever a subprocess is started, along with a set of arguments that will be passed to the callable. (Contributed by lekma; issue 5585.)

  The `Pool` class, which controls a pool of worker processes, now has an optional `maxtasksperchild` parameter. Worker processes will perform the specified number of tasks and then exit, causing the `Pool` to start a new worker. This is useful if tasks may leak memory or other resources, or if some tasks will cause the worker to become very large. (Contributed by Charles Cazabon; issue 6963.)

- The `nntplib` module now supports IPv6 addresses. (Contributed by Derek Morr; issue 1664.)

- New functions: the `os` module wraps the following POSIX system calls: `getresgid()` and `getresuid()`, which return the real, effective, and saved GIDs and UIDs; `setresgid()` and `setresuid()`, which set real, effective, and saved GIDs and UIDs to new values; `initgroups()`, which initialize the group access
list for the current process. (GID/UID functions contributed by Travis H.; issue 6508. Support for initgroups added by Jean-Paul Calderone; issue 7333.)

The `os.fork()` function now re-initializes the import lock in the child process; this fixes problems on Solaris when `fork()` is called from a thread. (Fixed by Zsolt Cserna; issue 7242.)

- In the `os.path` module, the `normpath()` and `abspath()` functions now preserve Unicode; if their input path is a Unicode string, the return value is also a Unicode string. (`normpath()` fixed by Matt Giuca in issue 5827; `abspath()` fixed by Ezio Melotti in issue 3426.)

- The `pydoc` module now has help for the various symbols that Python uses. You can now do `help('<<')` or `help('@')`, for example. (Contributed by David Laban; issue 4739.)

- The `re` module's `split()`, `sub()`, and `subn()` now accept an optional `flags` argument, for consistency with the other functions in the module. (Added by Gregory P. Smith.)

- New function: `run_path()` in the `runpy` module will execute the code at a provided `path` argument. `path` can be the path of a Python source file (`example.py`), a compiled bytecode file (`example.pyc`), a directory (`./package/`), or a zip archive (`example.zip`). If a directory or zip path is provided, it will be added to the front of `sys.path` and the module `__main__` will be imported. It's expected that the directory or zip contains a `__main__.py`; if it doesn't, some other `__main__.py` might be imported from a location later in `sys.path`. This makes more of the machinery of `runpy` available to scripts that want to mimic the way Python's command line processes an explicit path
name. (Added by Nick Coghlan; issue 6816.)

- New function: in the `shutil` module, `make_archive()` takes a filename, archive type (zip or tar-format), and a directory path, and creates an archive containing the directory’s contents. (Added by Tarek Ziadé.)

  `shutil`'s `copyfile()` and `copytree()` functions now raise a `SpecialFileError` exception when asked to copy a named pipe. Previously the code would treat named pipes like a regular file by opening them for reading, and this would block indefinitely. (Fixed by Antoine Pitrou; issue 3002.)

- The `signal` module no longer re-installs the signal handler unless this is truly necessary, which fixes a bug that could make it impossible to catch the EINTR signal robustly. (Fixed by Charles-Francois Natali; issue 8354.)

- New functions: in the `site` module, three new functions return various site- and user-specific paths. `getsitepackages()` returns a list containing all global site-packages directories, `getusersitepackages()` returns the path of the user’s site-packages directory, and `getuserbase()` returns the value of the `USER_BASE` environment variable, giving the path to a directory that can be used to store data. (Contributed by Tarek Ziadé; issue 6693.)

  The `site` module now reports exceptions occurring when the `sitecustomize` module is imported, and will no longer catch and swallow the `KeyboardInterrupt` exception. (Fixed by Victor Stinner; issue 3137.)

- The `create_connection()` function gained a `source_address` parameter, a `(host, port)` 2-tuple giving the source address
that will be used for the connection. (Contributed by Eldon Ziegler; issue 3972.)

The `recv_into()` and `recvfrom_into()` methods will now write into objects that support the buffer API, most usefully the `bytearray` and `memoryview` objects. (Implemented by Antoine Pitrou; issue 8104.)

- The `SocketServer` module’s `TCPServer` class now supports socket timeouts and disabling the Nagle algorithm. The `disable_nagle_algorithm` class attribute defaults to `False`; if overridden to be true, new request connections will have the TCP_NODELAY option set to prevent buffering many small sends into a single TCP packet. The `timeout` class attribute can hold a timeout in seconds that will be applied to the request socket; if no request is received within that time, `handle_timeout()` will be called and `handle_request()` will return. (Contributed by Kristján Valur Jónsson; issue 6192 and issue 6267.)

- Updated module: the `sqlite3` module has been updated to version 2.6.0 of the `pysqlite` package. Version 2.6.0 includes a number of bugfixes, and adds the ability to load SQLite extensions from shared libraries. Call the `enable_load_extension(True)` method to enable extensions, and then call `load_extension()` to load a particular shared library. (Updated by Gerhard Häring.)

- The `ssl` module’s `SSLSocket` objects now support the buffer API, which fixed a test suite failure (fix by Antoine Pitrou; issue 7133) and automatically set OpenSSL’s `SSL_MODE_AUTO_RETRY`, which will prevent an error code being returned from `recv()` operations that trigger an SSL renegotiation (fix by Antoine Pitrou; issue 8222).
The `ssl.wrap_socket()` constructor function now takes a `ciphers` argument that's a string listing the encryption algorithms to be allowed; the format of the string is described in the OpenSSL documentation. (Added by Antoine Pitrou; issue 8322.)

Another change makes the extension load all of OpenSSL's ciphers and digest algorithms so that they're all available. Some SSL certificates couldn't be verified, reporting an “unknown algorithm” error. (Reported by Beda Kosata, and fixed by Antoine Pitrou; issue 8484.)

The version of OpenSSL being used is now available as the module attributes `ssl.OPENSSL_VERSION` (a string), `ssl.OPENSSL_VERSION_INFO` (a 5-tuple), and `ssl.OPENSSL_VERSION_NUMBER` (an integer). (Added by Antoine Pitrou; issue 8321.)

- The `struct` module will no longer silently ignore overflow errors when a value is too large for a particular integer format code (one of `bBhHIlLqQ`); it now always raises a `struct.error` exception. (Changed by Mark Dickinson; issue 1523.) The `pack()` function will also attempt to use `__index__()` to convert and pack non-integers before trying the `__int__() method or reporting an error. (Changed by Mark Dickinson; issue 8300.)

- New function: the `subprocess` module’s `check_output()` runs a command with a specified set of arguments and returns the command’s output as a string when the command runs without error, or raises a `CalledProcessError` exception otherwise.

```
>>> subprocess.check_output(['df', '-h', '.'])
'Filesystem		Size		Used		Avail	Capacity		Mounted on
/dev/disk0s2		52G		49G		3.0G		94% /

>>> subprocess.check_output(['df', '-h', '/bogus'])
...
```
subprocess.CalledProcessError: Command ['df', '-h', '/bogus'] returned non-zero exit status 1

(Contributed by Gregory P. Smith.)

The subprocess module will now retry its internal system calls on receiving an EINTR signal. (Reported by several people; final patch by Gregory P. Smith in issue 1068268.)

- New function: is_declared_global() in the symtable module returns true for variables that are explicitly declared to be global, false for ones that are implicitly global. (Contributed by Jeremy Hylton.)

- The syslog module will now use the value of sys.argv[0] as the identifier instead of the previous default value of 'python'. (Changed by Sean Reifschneider; issue 8451.)

- The sys.version_info value is now a named tuple, with attributes named major, minor, micro, releaselevel, and serial. (Contributed by Ross Light; issue 4285.)

  sys.getwindowsversion() also returns a named tuple, with attributes named major, minor, build, platform, service_pack, service_pack_major, service_pack_minor, suite_mask, and product_type. (Contributed by Brian Curtin; issue 7766.)

- The tarfile module's default error handling has changed, to no longer suppress fatal errors. The default error level was previously 0, which meant that errors would only result in a message being written to the debug log, but because the debug log is not activated by default, these errors go unnoticed. The default error level is now 1, which raises an exception if there's an error. (Changed by Lars Gustäbel; issue 7357.)
**tarfile** now supports filtering the **TarInfo** objects being added to a tar file. When you call **add()**, you may supply an optional **filter** argument that's a callable. The **filter** callable will be passed the **TarInfo** for every file being added, and can modify and return it. If the callable returns **None**, the file will be excluded from the resulting archive. This is more powerful than the existing **exclude** argument, which has therefore been deprecated. (Added by Lars Gustäbel; issue 6856.) The **TarFile** class also now supports the context management protocol. (Added by Lars Gustäbel; issue 7232.)

- The **wait()** method of the **threading.Event** class now returns the internal flag on exit. This means the method will usually return true because **wait()** is supposed to block until the internal flag becomes true. The return value will only be false if a timeout was provided and the operation timed out. (Contributed by Tim Lesher; issue 1674032.)

- The Unicode database provided by the **unicodedata** module is now used internally to determine which characters are numeric, whitespace, or represent line breaks. The database also includes information from the **Unihan.txt** data file (patch by Anders Chrigström and Amaury Forgeot d'Arc; issue 1571184) and has been updated to version 5.2.0 (updated by Florent Xicluna; issue 8024).

- The **urlparse** module's **urlsplit()** now handles unknown URL schemes in a fashion compliant with **RFC 3986**: if the URL is of the form "<something>://...", the text before the :// is treated as the scheme, even if it's a made-up scheme that the module doesn't know about. This change may break code that worked around the old behaviour. For example, Python 2.6.4 or 2.5 will return the following:
```python
>>> import urlparse
>>> urlparse.urlsplit('invented://host/filename?query')
('invented', '', '//host/filename?query', '', '')
```

Python 2.7 (and Python 2.6.5) will return:

```python
>>> import urlparse
>>> urlparse.urlsplit('invented://host/filename?query')
('invented', 'host', '/filename?query', '', '')
```

(Python 2.7 actually produces slightly different output, since it returns a named tuple instead of a standard tuple.)

The `urlparse` module also supports IPv6 literal addresses as defined by RFC 2732 (contributed by Senthil Kumaran; issue 2987).

```python
>>> urlparse.urlparse('http://[1080::8:800:200C:417A]/foo')
ParseResult(scheme='http', netloc='[1080::8:800:200C:417A]', path='/foo', params='', query='', fragment='')
```

- New class: the `WeakSet` class in the `weakref` module is a set that only holds weak references to its elements; elements will be removed once there are no references pointing to them. (Originally implemented in Python 3.x by Raymond Hettinger, and backported to 2.7 by Michael Foord.)

- The ElementTree library, `xml.etree`, no longer escapes ampersands and angle brackets when outputting an XML processing instruction (which looks like `<?xml-stylesheet href="#style1"?>`) or comment (which looks like `<!-- comment -->`). (Patch by Neil Muller; issue 2746.)

- The XML-RPC client and server, provided by the `xmlrpclib` and `SimpleXMLRPCServer` modules, have improved performance by supporting HTTP/1.1 keep-alive and by optionally using gzip
encoding to compress the XML being exchanged. The gzip compression is controlled by the `encode_threshold` attribute of `SimpleXMLRPCRequestHandler`, which contains a size in bytes; responses larger than this will be compressed. (Contributed by Kristján Valur Jónsson; issue 6267.)

- The `zipfile` module’s `ZipFile` now supports the context management protocol, so you can write `with zipfile.ZipFile(...) as f:`. (Contributed by Brian Curtin; issue 5511.)

  `zipfile` now also supports archiving empty directories and extracts them correctly. (Fixed by Kuba Wieczorek; issue 4710.) Reading files out of an archive is faster, and interleaving `read()` and `readline()` now works correctly. (Contributed by Nir Aides; issue 7610.)

  The `is_zipfile()` function now accepts a file object, in addition to the path names accepted in earlier versions. (Contributed by Gabriel Genellina; issue 4756.)

  The `writestr()` method now has an optional `compress_type` parameter that lets you override the default compression method specified in the `ZipFile` constructor. (Contributed by Ronald Oussoren; issue 6003.)

New module: `importlib`

Python 3.1 includes the `importlib` package, a re-implementation of the logic underlying Python’s `import` statement. `importlib` is useful for implementors of Python interpreters and to users who wish to write new importers that can participate in the import process. Python 2.7 doesn’t contain the complete `importlib` package, but
instead has a tiny subset that contains a single function, `import_module()`.

`import_module(name, package=None)` imports a module. `name` is a string containing the module or package's name. It's possible to do relative imports by providing a string that begins with a . character, such as `..utils.errors`. For relative imports, the `package` argument must be provided and is the name of the package that will be used as the anchor for the relative import. `import_module()` both inserts the imported module into `sys.modules` and returns the module object.

Here are some examples:

```python
>>> from importlib import import_module
>>> anydbm = import_module('anydbm')  # Standard absolute import
>>> anydbm
<module 'anydbm' from '/p/python/Lib/anydbm.py'>
>>> # Relative import
>>> file_util = import_module('..file_util', 'distutils.command')
>>> file_util
<module 'distutils.file_util' from '/python/Lib/distutils/file_util.pyc'>
```

`importlib` was implemented by Brett Cannon and introduced in Python 3.1.

### New module: sysconfig

The `sysconfig` module has been pulled out of the Distutils package, becoming a new top-level module in its own right. `sysconfig` provides functions for getting information about Python's build process: compiler switches, installation paths, the platform name, and whether Python is running from its source directory.

Some of the functions in the module are:
• `get_config_var()` returns variables from Python’s Makefile and the `pyconfig.h` file.

• `get_config_vars()` returns a dictionary containing all of the configuration variables.

• `get_path()` returns the configured path for a particular type of module: the standard library, site-specific modules, platform-specific modules, etc.

• `is_python_build()` returns true if you’re running a binary from a Python source tree, and false otherwise.

Consult the `sysconfig` documentation for more details and for a complete list of functions.

The Distutils package and `sysconfig` are now maintained by Tarek Ziadé, who has also started a Distutils2 package (source repository at https://hg.python.org/distutils2/) for developing a next-generation version of Distutils.

**ttk: Themed Widgets for Tk**

Tcl/Tk 8.5 includes a set of themed widgets that re-implement basic Tk widgets but have a more customizable appearance and can therefore more closely resemble the native platform’s widgets. This widget set was originally called Tile, but was renamed to Ttk (for “themed Tk”) on being added to Tcl/Tk release 8.5.

To learn more, read the `ttk` module documentation. You may also wish to read the Tcl/Tk manual page describing the Ttk theme engine, available at https://www.tcl.tk/man/tcl8.5/TkCmd/ttk_intro.htm. Some screenshots of the Python/Ttk code in use are at http://code.google.com/p/python-ttk/wiki/Screenshots.

The `ttk` module was written by Guilherme Polo and added in issue
An alternate version called Tile.py, written by Martin Franklin and maintained by Kevin Walzer, was proposed for inclusion in issue 2618, but the authors argued that Guilherme Polo’s work was more comprehensive.

**Updated module: unittest**

The unittest module was greatly enhanced; many new features were added. Most of these features were implemented by Michael Foord, unless otherwise noted. The enhanced version of the module is downloadable separately for use with Python versions 2.4 to 2.6, packaged as the unittest2 package, from https://pypi.python.org/pypi/unittest2.

When used from the command line, the module can automatically discover tests. It’s not as fancy as py.test or nose, but provides a simple way to run tests kept within a set of package directories. For example, the following command will search the test/ subdirectory for any importable test files named test*.py:

```
python -m unittest discover -s test
```

Consult the unittest module documentation for more details. (Developed in issue 6001.)

The main() function supports some other new options:

- `-b` or `--buffer` will buffer the standard output and standard error streams during each test. If the test passes, any resulting output will be discarded; on failure, the buffered output will be displayed.

- `-c` or `--catch` will cause the control-C interrupt to be handled more gracefully. Instead of interrupting the test process
immediately, the currently running test will be completed and then the partial results up to the interruption will be reported. If you’re impatient, a second press of control-C will cause an immediate interruption.

This control-C handler tries to avoid causing problems when the code being tested or the tests being run have defined a signal handler of their own, by noticing that a signal handler was already set and calling it. If this doesn’t work for you, there’s a removeHandler() decorator that can be used to mark tests that should have the control-C handling disabled.

- `-f` or `--failfast` makes test execution stop immediately when a test fails instead of continuing to execute further tests. (Suggested by Cliff Dyer and implemented by Michael Foord; issue 8074.)

The progress messages now show ‘x’ for expected failures and ‘u’ for unexpected successes when run in verbose mode. (Contributed by Benjamin Peterson.)

Test cases can raise the `SkipTest` exception to skip a test (issue 1034053).

The error messages for `assertTrue()`, `assertFalse()`, and `assertEqual()` failures now provide more information. If you set the `longMessage` attribute of your `TestCase` classes to true, both the standard error message and any additional message you provide will be printed for failures. (Added by Michael Foord; issue 5663.)

The `assertRaises()` method now returns a context handler when called without providing a callable object to run. For example, you can write this:

```python
with self.assertRaises(KeyError):
```
Module- and class-level setup and teardown fixtures are now supported. Modules can contain `setUpModule()` and `tearDownModule()` functions. Classes can have `setUpClass()` and `tearDownClass()` methods that must be defined as class methods (using `@classmethod` or equivalent). These functions and methods are invoked when the test runner switches to a test case in a different module or class.

The methods `addCleanup()` and `doCleanups()` were added. `addCleanup()` lets you add cleanup functions that will be called unconditionally (after `setUp()` if `setUp()` fails, otherwise after `tearDown()`). This allows for much simpler resource allocation and deallocation during tests (issue 5679).

A number of new methods were added that provide more specialized tests. Many of these methods were written by Google engineers for use in their test suites; Gregory P. Smith, Michael Foord, and GvR worked on merging them into Python’s version of `unittest`.

- `assertIsNone()` and `assertIsNotNone()` take one expression and verify that the result is or is not `None`.
- `assertIs()` and `assertIsNot()` take two values and check whether the two values evaluate to the same object or not. (Added by Michael Foord; issue 2578.)
- `assertIsInstance()` and `assertNotIsInstance()` check whether the resulting object is an instance of a particular class, or of one of a tuple of classes. (Added by Georg Brandl; issue 7031.)
- `assertGreater()`, `assertGreaterEqual()`, `assertLess()`, and `assertLessEqual()` compare two quantities.
- `assertMultiLineEqual()` compares two strings, and if they're not equal, displays a helpful comparison that highlights the differences in the two strings. This comparison is now used by default when Unicode strings are compared with `assertEqual()`.
- `assertRegexpMatches()` and `assertNotRegexpMatches()` checks whether the first argument is a string matching or not matching the regular expression provided as the second argument (issue 8038).
- `assertRaisesRegexp()` checks whether a particular exception is raised, and then also checks that the string representation of the exception matches the provided regular expression.
- `assertIn()` and `assertNotIn()` tests whether `first` is or is not in `second`.
- `assertItemsEqual()` tests whether two provided sequences contain the same elements.
- `assertSetEqual()` compares whether two sets are equal, and only reports the differences between the sets in case of error.
- Similarly, `assertListEqual()` and `assertTupleEqual()` compare the specified types and explain any differences without necessarily printing their full values; these methods are now used by default when comparing lists and tuples using `assertEqual()`. More generally, `assertSequenceEqual()` compares two sequences and can optionally check whether both sequences are of a particular type.
- `assertDictEqual()` compares two dictionaries and reports the differences; it's now used by default when you compare two dictionaries using `assertEqual()`. `assertDictContainsSubset()` checks whether all of the key/value pairs in `first` are found in `second`.
- `assertAlmostEqual()` and `assertNotAlmostEqual()` test whether `first` and `second` are approximately equal. This method can either round their difference to an optionally-specified number of places (the default is 7) and compare it to zero, or require the
difference to be smaller than a supplied \textit{delta} value.

- \texttt{loadTestsFromName()} properly honors the \texttt{suiteClass} attribute of the \texttt{TestLoader}. (Fixed by Mark Roddy; \texttt{issue 6866}.)

- A new hook lets you extend the \texttt{assertEqual()} method to handle new data types. The \texttt{addTypeEqualityFunc()} method takes a type object and a function. The function will be used when both of the objects being compared are of the specified type. This function should compare the two objects and raise an exception if they don't match; it's a good idea for the function to provide additional information about why the two objects aren't matching, much as the new sequence comparison methods do.

\texttt{unittest.main()} now takes an optional \texttt{exit} argument. If false, \texttt{main()} doesn't call \texttt{sys.exit()}, allowing \texttt{main()} to be used from the interactive interpreter. (Contributed by J. Pablo Fernández; \texttt{issue 3379}.)

\texttt{TestResult} has new \texttt{startTestRun()} and \texttt{stopTestRun()} methods that are called immediately before and after a test run. (Contributed by Robert Collins; \texttt{issue 5728}.)

With all these changes, the \texttt{unittest.py} was becoming awkwardly large, so the module was turned into a package and the code split into several files (by Benjamin Peterson). This doesn't affect how the module is imported or used.

\textbf{See also:}

\url{http://www.voidspace.org.uk/python/articles/unittest2.shtml}

Describes the new features, how to use them, and the rationale for various design decisions. (By Michael Foord.)

\textbf{Updated module: ElementTree 1.3}
The version of the ElementTree library included with Python was updated to version 1.3. Some of the new features are:

- The various parsing functions now take a parser keyword argument giving an XMLParser instance that will be used. This makes it possible to override the file’s internal encoding:

  ```python
  p = ET.XMLParser(encoding='utf-8')
  t = ET.XML('""<root/>""', parser=p)
  ```

Errors in parsing XML now raise a ParseError exception, whose instances have a position attribute containing a (line, column) tuple giving the location of the problem.

- ElementTree’s code for converting trees to a string has been significantly reworked, making it roughly twice as fast in many cases. The ElementTree.write() and Element.write() methods now have a method parameter that can be “xml” (the default), “html”, or “text”. HTML mode will output empty elements as <empty></empty> instead of <empty/>; and text mode will skip over elements and only output the text chunks. If you set the tag attribute of an element to None but leave its children in place, the element will be omitted when the tree is written out, so you don’t need to do more extensive rearrangement to remove a single element.

Namespace handling has also been improved. All xmlns:<whatever> declarations are now output on the root element, not scattered throughout the resulting XML. You can set the default namespace for a tree by setting the default_namespace attribute and can register new prefixes with register_namespace(). In XML mode, you can use the true/false xml_declaration parameter to suppress the XML declaration.
• New **Element** method: `extend()` appends the items from a sequence to the element’s children. Elements themselves behave like sequences, so it’s easy to move children from one element to another:

```python
from xml.etree import ElementTree as ET

t = ET.XML('""<list>
    <item>1</item> <item>2</item> <item>3</item>
</list>"
)new = ET.XML('<root/>'
new.extend(t)
# Outputs <root><item>1</item>...</root>
print ET.tostring(new)
```

• New **Element** method: `iter()` yields the children of the element as a generator. It’s also possible to write `for child in elem:` to loop over an element’s children. The existing method `getiterator()` is now deprecated, as is `getchildren()` which constructs and returns a list of children.

• New **Element** method: `itertext()` yields all chunks of text that are descendants of the element. For example:

```python
t = ET.XML('""<list>
    <item>1</item> <item>2</item> <item>3</item>
</list>"
)
# Outputs ['
    ', '1', ' ', '2', ' ', '3', ' \n']
print list(t.itertext())
```

• Deprecated: using an element as a Boolean (i.e., `if elem:`) would return true if the element had any children, or false if there were no children. This behaviour is confusing – `None` is false, but so is a childless element? – so it will now trigger a `FutureWarning`. In your code, you should be explicit: write `len(elem) != 0` if you’re interested in the number of children, or
elem is not None.

Fredrik Lundh develops ElementTree and produced the 1.3 version; you can read his article describing 1.3 at http://effbot.org/zone/elementtree-13-intro.htm. Florent Xicluna updated the version included with Python, after discussions on python-dev and in issue 6472.)
Build and C API Changes

Changes to Python’s build process and to the C API include:

- The latest release of the GNU Debugger, GDB 7, can be scripted using Python. When you begin debugging an executable program P, GDB will look for a file named \texttt{P-gdb.py} and automatically read it. Dave Malcolm contributed a \texttt{python-gdb.py} that adds a number of commands useful when debugging Python itself. For example, \texttt{py-up} and \texttt{py-down} go up or down one Python stack frame, which usually corresponds to several C stack frames. \texttt{py-print} prints the value of a Python variable, and \texttt{py-bt} prints the Python stack trace. (Added as a result of issue 8032.)

- If you use the \texttt{.gdbinit} file provided with Python, the “pyo” macro in the 2.7 version now works correctly when the thread being debugged doesn’t hold the GIL; the macro now acquires it before printing. (Contributed by Victor Stinner; issue 3632.)

- \texttt{Py_AddPendingCall()} is now thread-safe, letting any worker thread submit notifications to the main Python thread. This is particularly useful for asynchronous IO operations. (Contributed by Kristján Valur Jónsson; issue 4293.)

- New function: \texttt{PyCode_NewEmpty()} creates an empty code object; only the filename, function name, and first line number are required. This is useful for extension modules that are attempting to construct a more useful traceback stack. Previously such extensions needed to call \texttt{PyCode_New()}, which had many more arguments. (Added by Jeffrey Yasskin.)

- New function: \texttt{PyErr_NewExceptionWithDoc()} creates a new
exception class, just as the existing `PyErr_NewException()` does, but takes an extra `char *` argument containing the docstring for the new exception class. (Added by ‘lekma’ on the Python bug tracker; issue 7033.)

- New function: `PyFrame_GetLineNumber()` takes a frame object and returns the line number that the frame is currently executing. Previously code would need to get the index of the bytecode instruction currently executing, and then look up the line number corresponding to that address. (Added by Jeffrey Yasskin.)

- New functions: `PyLong_AsLongAndOverflow()` and `PyLong_AsLongLongAndOverflow()` approximates a Python long integer as a C long or long long. If the number is too large to fit into the output type, an `overflow` flag is set and returned to the caller. (Contributed by Case Van Horsen; issue 7528 and issue 7767.)

- New function: stemming from the rewrite of string-to-float conversion, a new `PyOS_string_to_double()` function was added. The old `PyOS_ascii_strtod()` and `PyOS_ascii_atof()` functions are now deprecated.

- New function: `PySys_SetArgvEx()` sets the value of `sys.argv` and can optionally update `sys.path` to include the directory containing the script named by `sys.argv[0]` depending on the value of an `updatepath` parameter.

This function was added to close a security hole for applications that embed Python. The old function, `PySys_SetArgv()`, would always update `sys.path`, and sometimes it would add the current directory. This meant that, if you ran an application embedding Python in a directory controlled by someone else,
attackers could put a Trojan-horse module in the directory (say, a file named `os.py`) that your application would then import and run.

If you maintain a C/C++ application that embeds Python, check whether you’re calling `PySys_SetArgv()` and carefully consider whether the application should be using `PySys_SetArgvEx()` with `updatepath` set to false.

Security issue reported as CVE-2008-5983; discussed in issue 5753, and fixed by Antoine Pitrou.

- New macros: the Python header files now define the following macros: `Py_ISALNUM`, `Py_ISALPHA`, `Py_ISDIGIT`, `Py_ISLOWER`, `Py_ISSPACE`, `Py_ISUPPER`, `Py_ISXDIGIT`, `Py_TOLOWER`, and `Py_TOUPPER`. All of these functions are analogous to the C standard macros for classifying characters, but ignore the current locale setting, because in several places Python needs to analyze characters in a locale-independent way. (Added by Eric Smith; issue 5793.)

- Removed function: `PyEval_CallObject` is now only available as a macro. A function version was being kept around to preserve ABI linking compatibility, but that was in 1997; it can certainly be deleted by now. (Removed by Antoine Pitrou; issue 8276.)

- New format codes: the `PyFormat_FromString()`, `PyFormat_FromStringV()`, and `PyErr_Format()` functions now accept `%lld` and `%llu` format codes for displaying C’s `long long` types. (Contributed by Mark Dickinson; issue 7228.)

- The complicated interaction between threads and process forking has been changed. Previously, the child process created by `os.fork()` might fail because the child is created with only a
single thread running, the thread performing the \texttt{os.fork()}. If other threads were holding a lock, such as Python's import lock, when the fork was performed, the lock would still be marked as "held" in the new process. But in the child process nothing would ever release the lock, since the other threads weren't replicated, and the child process would no longer be able to perform imports.

Python 2.7 acquires the import lock before performing an \texttt{os.fork()}, and will also clean up any locks created using the \texttt{threading} module. C extension modules that have internal locks, or that call \texttt{fork()} themselves, will not benefit from this clean-up.

(Fixed by Thomas Wouters; issue 1590864.)

- The \texttt{Py_Finalize()} function now calls the internal \texttt{threading._shutdown()} function; this prevents some exceptions from being raised when an interpreter shuts down. (Patch by Adam Olsen; issue 1722344.)

- When using the \texttt{PyMemberDef} structure to define attributes of a type, Python will no longer let you try to delete or set a \texttt{T\_STRING\_INPLACE} attribute.

- Global symbols defined by the \texttt{ctypes} module are now prefixed with \texttt{Py}, or with \texttt{_ctypes}. (Implemented by Thomas Heller; issue 3102.)

- New configure option: the \texttt{--with-system-expat} switch allows building the \texttt{pyexpat} module to use the system Expat library. (Contributed by Arfrever Frehtes Taifersar Arahesis; issue 7609.)
- New configure option: the `--with-valgrind` option will now disable the pymalloc allocator, which is difficult for the Valgrind memory-error detector to analyze correctly. Valgrind will therefore be better at detecting memory leaks and overruns. (Contributed by James Henstridge; issue 2422.)

- New configure option: you can now supply an empty string to `--with-dbmliborder=` in order to disable all of the various DBM modules. (Added by Arfrever Frehtes Taifersar Arahesis; issue 6491.)

- The `configure` script now checks for floating-point rounding bugs on certain 32-bit Intel chips and defines a `X87_DOUBLE_ROUNDING` preprocessor definition. No code currently uses this definition, but it's available if anyone wishes to use it. (Added by Mark Dickinson; issue 2937.)

  `configure` also now sets a `LDCXXSHARED` Makefile variable for supporting C++ linking. (Contributed by Arfrever Frehtes Taifersar Arahesis; issue 1222585.)

- The build process now creates the necessary files for pkg-config support. (Contributed by Clinton Roy; issue 3585.)

- The build process now supports Subversion 1.7. (Contributed by Arfrever Frehtes Taifersar Arahesis; issue 6094.)

### Capsules

Python 3.1 adds a new C datatype, `PyCapsule`, for providing a C API to an extension module. A capsule is essentially the holder of a C `void *` pointer, and is made available as a module attribute; for example, the `socket` module's API is exposed as `socket.CAPI`, and `unicodedata` exposes `ucnhash_CAPI`. Other extensions can import the
module, access its dictionary to get the capsule object, and then get the `void *` pointer, which will usually point to an array of pointers to the module's various API functions.

There is an existing data type already used for this, `PyCObject`, but it doesn’t provide type safety. Evil code written in pure Python could cause a segmentation fault by taking a `PyCObject` from module A and somehow substituting it for the `PyCObject` in module B. Capsules know their own name, and getting the pointer requires providing the name:

```python
void *vtable;

if (!PyCapsule_IsValid(capsule, "mymodule.CAPI") {
    PyErr_SetString(PyExc_ValueError, "argument type invalid");
    return NULL;
}

vtable = PyCapsule_GetPointer(capsule, "mymodule.CAPI");
```

You are assured that `vtable` points to whatever you’re expecting. If a different capsule was passed in, `PyCapsule_IsValid()` would detect the mismatched name and return false. Refer to Providing a C API for an Extension Module for more information on using these objects.

Python 2.7 now uses capsules internally to provide various extension-module APIs, but the `PyCObject_AsVoidPtr()` was modified to handle capsules, preserving compile-time compatibility with the `cObject` interface. Use of `PyCObject_AsVoidPtr()` will signal a `PendingDeprecationWarning`, which is silent by default.

Implemented in Python 3.1 and backported to 2.7 by Larry Hastings; discussed in issue 5630.

### Port-Specific Changes: Windows
- The `msvcrt` module now contains some constants from the `crtassem.h` header file: `CRT_ASSEMBLY_VERSION`, `VC_ASSEMBLY_PUBLICKEYTOKEN`, and `LIBRARIES_ASSEMBLY_NAME_PREFIX`. (Contributed by David Cournapeau; issue 4365.)

- The `_winreg` module for accessing the registry now implements the `CreateKeyEx()` and `DeleteKeyEx()` functions, extended versions of previously-supported functions that take several extra arguments. The `DisableReflectionKey()`, `EnableReflectionKey()`, and `QueryReflectionKey()` were also tested and documented. (Implemented by Brian Curtin: issue 7347.)

- The new `_beginthreadex()` API is used to start threads, and the native thread-local storage functions are now used. (Contributed by Kristján Valur Jónsson; issue 3582.)

- The `os.kill()` function now works on Windows. The signal value can be the constants `CTRL_C_EVENT`, `CTRL_BREAK_EVENT`, or any integer. The first two constants will send Control-C and Control-Break keystroke events to subprocesses; any other value will use the `TerminateProcess()` API. (Contributed by Miki Tebeka; issue 1220212.)

- The `os.listdir()` function now correctly fails for an empty path. (Fixed by Hirokazu Yamamoto; issue 5913.)

- The `mimelib` module will now read the MIME database from the Windows registry when initializing. (Patch by Gabriel Genellina; issue 4969.)

---

**Port-Specific Changes: Mac OS X**

- The path `/Library/Python/2.7/site-packages` is now appended to `sys.path`, in order to share added packages between the system installation and a user-installed copy of the same
version. (Changed by Ronald Oussoren; issue 4865.)

*Changed in version 2.7.13:* As of 2.7.13, this change was removed. /Library/Python/2.7/site-packages, the site-packages directory used by the Apple-supplied system Python 2.7 is no longer appended to `sys.path` for user-installed Pythons such as from the python.org installers. As of macOS 10.12, Apple changed how the system site-packages directory is configured, which could cause installation of pip components, like setuptools, to fail. Packages installed for the system Python will no longer be shared with user-installed Pythons. (issue 28440)

**Port-Specific Changes: FreeBSD**

- FreeBSD 7.1’s `SO_SETFIB` constant, used with `getsockopt()`/`setsockopt()` to select an alternate routing table, is now available in the `socket` module. (Added by Kyle VanderBeek; issue 8235.)
Other Changes and Fixes

- Two benchmark scripts, `iobench` and `ccbench`, were added to the `Tools` directory. `iobench` measures the speed of the built-in file I/O objects returned by `open()` while performing various operations, and `ccbench` is a concurrency benchmark that tries to measure computing throughput, thread switching latency, and IO processing bandwidth when performing several tasks using a varying number of threads.

- The `Tools/i18n/msgfmt.py` script now understands plural forms in `.po` files. (Fixed by Martin von Löwis; issue 5464.)

- When importing a module from a `.pyc` or `.pyo` file with an existing `.py` counterpart, the `co_filename` attributes of the resulting code objects are overwritten when the original filename is obsolete. This can happen if the file has been renamed, moved, or is accessed through different paths. (Patch by Ziga Seilnacht and Jean-Paul Calderone; issue 1180193.)

- The `regrtest.py` script now takes a `--randseed=` switch that takes an integer that will be used as the random seed for the `-r` option that executes tests in random order. The `-r` option also reports the seed that was used (Added by Collin Winter.)

- Another `regrtest.py` switch is `-j`, which takes an integer specifying how many tests run in parallel. This allows reducing the total runtime on multi-core machines. This option is compatible with several other options, including the `-R` switch which is known to produce long runtimes. (Added by Antoine Pitrou, issue 6152.) This can also be used with a new `-F` switch that runs selected tests in a loop until they fail. (Added by Antoine Pitrou; issue 7312.)

- When executed as a script, the `py_compile.py` module now accepts `'-` as an argument, which will read standard input for
the list of filenames to be compiled. (Contributed by Piotr Ożarowski; issue 8233.)
Porting to Python 2.7

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

- The `range()` function processes its arguments more consistently; it will now call `__int__() on non-float, non-integer arguments that are supplied to it. (Fixed by Alexander Belopolsky; issue 1533.)
- The string `format()` method changed the default precision used for floating-point and complex numbers from 6 decimal places to 12, which matches the precision used by `str()`. (Changed by Eric Smith; issue 5920.)
- Because of an optimization for the `with` statement, the special methods `__enter__()` and `__exit__()` must belong to the object’s type, and cannot be directly attached to the object’s instance. This affects new-style classes (derived from `object`) and C extension types. (issue 6101.)
- Due to a bug in Python 2.6, the `exc_value` parameter to `__exit__()` methods was often the string representation of the exception, not an instance. This was fixed in 2.7, so `exc_value` will be an instance as expected. (Fixed by Florent Xicluna; issue 7853.)
- When a restricted set of attributes were set using `__slots__`, deleting an unset attribute would not raise `AttributeError` as you would expect. Fixed by Benjamin Peterson; issue 7604.)

In the standard library:

- Operations with `datetime` instances that resulted in a year falling outside the supported range didn’t always raise `OverflowError`. Such errors are now checked more carefully and will now raise
the exception. (Reported by Mark Leander, patch by Anand B. Pillai and Alexander Belopolsky; issue 7150.)

- When using `Decimal` instances with a string's `format()` method, the default alignment was previously left-alignment. This has been changed to right-alignment, which might change the output of your programs. (Changed by Mark Dickinson; issue 6857.)

Comparisons involving a signaling NaN value (or `sNAN`) now signal `InvalidOperation` instead of silently returning a true or false value depending on the comparison operator. Quiet NaN values (or `NaN`) are now hashable. (Fixed by Mark Dickinson; issue 7279.)

- The ElementTree library, `xml.etree`, no longer escapes ampersands and angle brackets when outputting an XML processing instruction (which looks like `<?xml-stylesheet href="#style1"?>`) or comment (which looks like `<!– comment – >`). (Patch by Neil Muller; issue 2746.)

- The `readline()` method of `StringIO` objects now does nothing when a negative length is requested, as other file-like objects do. (issue 7348).

- The `syslog` module will now use the value of `sys.argv[0]` as the identifier instead of the previous default value of `'python'`. (Changed by Sean Reifschneider; issue 8451.)

- The `tarfile` module’s default error handling has changed, to no longer suppress fatal errors. The default error level was previously 0, which meant that errors would only result in a message being written to the debug log, but because the debug log is not activated by default, these errors go unnoticed. The default error level is now 1, which raises an exception if there's
an error. (Changed by Lars Gustäbel; issue 7357.)

- The **urlparse** module's **urlsplit()** now handles unknown URL schemes in a fashion compliant with [RFC 3986](https://tools.ietf.org/html/rfc3986): if the URL is of the form "<something>://...", the text before the :// is treated as the scheme, even if it's a made-up scheme that the module doesn't know about. This change may break code that worked around the old behaviour. For example, Python 2.6.4 or 2.5 will return the following:

```python
>>> import urlparse
>>> urlparse.urlsplit('invented://host/filename?query')
('invented', '', '//host/filename?query', '', '')
```

Python 2.7 (and Python 2.6.5) will return:

```python
>>> import urlparse
>>> urlparse.urlsplit('invented://host/filename?query')
('invented', 'host', '/filename?query', '', '')
```

(Python 2.7 actually produces slightly different output, since it returns a named tuple instead of a standard tuple.)

For C extensions:

- C extensions that use integer format codes with the **PyArg_Parse** family of functions will now raise a **TypeError** exception instead of triggering a **DeprecationWarning** (issue 5080).
- Use the new **PyOS_string_to_double()** function instead of the old **PyOS_ascii_strtod()** and **PyOS_ascii_atof()** functions, which are now deprecated.

For applications that embed Python:

- The **PySys_SetArgvEx()** function was added, letting applications
close a security hole when the existing `PySys_SetArgv()` function was used. Check whether you're calling `PySys_SetArgv()` and carefully consider whether the application should be using `PySys_SetArgvEx()` with `updatepath` set to false.
New Features Added to Python 2.7 Maintenance Releases

New features may be added to Python 2.7 maintenance releases when the situation genuinely calls for it. Any such additions must go through the Python Enhancement Proposal process, and make a compelling case for why they can't be adequately addressed by either adding the new feature solely to Python 3, or else by publishing it on the Python Package Index.

In addition to the specific proposals listed below, there is a general exemption allowing new `-3` warnings to be added in any Python 2.7 maintenance release.

**PEP 434: IDLE Enhancement Exception for All Branches**

**PEP 434** describes a general exemption for changes made to the IDLE development environment shipped along with Python. This exemption makes it possible for the IDLE developers to provide a more consistent user experience across all supported versions of Python 2 and 3.

For details of any IDLE changes, refer to the NEWS file for the specific release.

**PEP 466: Network Security Enhancements for Python 2.7**

**PEP 466** describes a number of network security enhancement proposals that have been approved for inclusion in Python 2.7 maintenance releases, with the first of those changes appearing in
the Python 2.7.7 release.

**PEP 466** related features added in Python 2.7.7:

- `hmac.compare_digest()` was backported from Python 3 to make a timing attack resistant comparison operation available to Python 2 applications. (Contributed by Alex Gaynor; issue 21306.)
- OpenSSL 1.0.1g was upgraded in the official Windows installers published on python.org. (Contributed by Zachary Ware; issue 21462.)

**PEP 466** related features added in Python 2.7.8:

- `hashlib.pbkdf2_hmac()` was backported from Python 3 to make a hashing algorithm suitable for secure password storage broadly available to Python 2 applications. (Contributed by Alex Gaynor; issue 21304.)
- OpenSSL 1.0.1h was upgraded for the official Windows installers published on python.org. (contributed by Zachary Ware in issue 21671 for CVE-2014-0224)

**PEP 466** related features added in Python 2.7.9:

- Most of Python 3.4’s `ssl` module was backported. This means `ssl` now supports Server Name Indication, TLS1.x settings, access to the platform certificate store, the `SSLContext` class, and other features. (Contributed by Alex Gaynor and David Reid; issue 21308.)

  Refer to the “Version added: 2.7.9” notes in the module documentation for specific details.

- `os.urandom()` was changed to cache a file descriptor to `/dev/urandom instead of reopening `/dev/urandom on every call.
Contributed by Alex Gaynor; issue 21305.

- `hashlib.algorithms_guaranteed` and `hashlib.algorithms_available` were backported from Python 3 to make it easier for Python 2 applications to select the strongest available hash algorithm. (Contributed by Alex Gaynor in issue 21307)

### PEP 477: Backport ensurepip (PEP 453) to Python 2.7

PEP 477 approves the inclusion of the PEP 453 ensurepip module and the improved documentation that was enabled by it in the Python 2.7 maintenance releases, appearing first in the Python 2.7.9 release.

### Bootstrapping pip By Default

The new `ensurepip` module (defined in PEP 453) provides a standard cross-platform mechanism to bootstrap the pip installer into Python installations. The version of `pip` included with Python 2.7.9 is pip 1.5.6, and future 2.7.x maintenance releases will update the bundled version to the latest version of `pip` that is available at the time of creating the release candidate.

By default, the commands `pip`, `pipX` and `pipX.Y` will be installed on all platforms (where X.Y stands for the version of the Python installation), along with the `pip` Python package and its dependencies.

For CPython source builds on POSIX systems, the `make install` and `make altinstall` commands do not bootstrap `pip` by default. This behaviour can be controlled through configure options, and
overridden through Makefile options.

On Windows and Mac OS X, the CPython installers now default to installing `pip` along with CPython itself (users may opt out of installing it during the installation process). Window users will need to opt in to the automatic `PATH` modifications to have `pip` available from the command line by default, otherwise it can still be accessed through the Python launcher for Windows as `py -m pip`.

As discussed in the PEP, platform packagers may choose not to install these commands by default, as long as, when invoked, they provide clear and simple directions on how to install them on that platform (usually using the system package manager).

**Documentation Changes**

As part of this change, the Installing Python Modules and Distributing Python Modules sections of the documentation have been completely redesigned as short getting started and FAQ documents. Most packaging documentation has now been moved out to the Python Packaging Authority maintained Python Packaging User Guide and the documentation of the individual projects.

However, as this migration is currently still incomplete, the legacy versions of those guides remaining available as Installing Python Modules (Legacy version) and Distributing Python Modules (Legacy version).

**See also:**

**PEP 453 – Explicit bootstrapping of pip in Python installations**

PEP written by Donald Stufft and Nick Coghlan, implemented by Donald Stufft, Nick Coghlan, Martin von Löwis and Ned Deily.
PEP 476: Enabling certificate verification by default for stdlib http clients

PEP 476 updated `httplib` and modules which use it, such as `urllib2` and `xmlrpclib`, to now verify that the server presents a certificate which is signed by a Certificate Authority in the platform trust store and whose hostname matches the hostname being requested by default, significantly improving security for many applications. This change was made in the Python 2.7.9 release.

For applications which require the old previous behavior, they can pass an alternate context:

```python
import urllib2
import ssl

# This disables all verification
context = ssl._create_unverified_context()

# This allows using a specific certificate for the host, which
# to be in the trust store
context = ssl.create_default_context(cafile='/path/to/file.crt')

urllib2.urlopen("https://invalid-cert", context=context)
```

PEP 493: HTTPS verification migration tools for Python 2.7

PEP 493 provides additional migration tools to support a more incremental infrastructure upgrade process for environments containing applications and services relying on the historically permissive processing of server certificates when establishing client HTTPS connections. These additions were made in the Python 2.7.12 release.

These tools are intended for use in cases where affected
applications and services can’t be modified to explicitly pass a more permissive SSL context when establishing the connection.

For applications and services which can’t be modified at all, the new PYTHONHTTPSVERIFY environment variable may be set to 0 to revert an entire Python process back to the default permissive behaviour of Python 2.7.8 and earlier.

For cases where the connection establishment code can’t be modified, but the overall application can be, the new ssl._https_verify_certificates() function can be used to adjust the default behaviour at runtime.
Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Nick Coghlan, Philip Jenvey, Ryan Lovett, R. David Murray, Hugh Secker-Walker.
What’s New in Python 2.6

Author: A.M. Kuchling (amk at amk.ca)

This article explains the new features in Python 2.6, released on October 1 2008. The release schedule is described in PEP 361.

The major theme of Python 2.6 is preparing the migration path to Python 3.0, a major redesign of the language. Whenever possible, Python 2.6 incorporates new features and syntax from 3.0 while remaining compatible with existing code by not removing older features or syntax. When it’s not possible to do that, Python 2.6 tries to do what it can, adding compatibility functions in a future_builtins module and a -3 switch to warn about usages that will become unsupported in 3.0.

Some significant new packages have been added to the standard library, such as the multiprocessing and json modules, but there aren’t many new features that aren’t related to Python 3.0 in some way.

Python 2.6 also sees a number of improvements and bugfixes throughout the source. A search through the change logs finds there were 259 patches applied and 612 bugs fixed between Python 2.5 and 2.6. Both figures are likely to be underestimates.

This article doesn’t attempt to provide a complete specification of the new features, but instead provides a convenient overview. For full details, you should refer to the documentation for Python 2.6. If you want to understand the rationale for the design and implementation, refer to the PEP for a particular new feature. Whenever possible, “What's New in Python” links to the bug/patch item for each change.
Python 3.0

The development cycle for Python versions 2.6 and 3.0 was synchronized, with the alpha and beta releases for both versions being made on the same days. The development of 3.0 has influenced many features in 2.6.

Python 3.0 is a far-ranging redesign of Python that breaks compatibility with the 2.x series. This means that existing Python code will need some conversion in order to run on Python 3.0. However, not all the changes in 3.0 necessarily break compatibility. In cases where new features won’t cause existing code to break, they’ve been backported to 2.6 and are described in this document in the appropriate place. Some of the 3.0-derived features are:

- A `__complex__()` method for converting objects to a complex number.
- Alternate syntax for catching exceptions: `except TypeError as exc`.
- The addition of `functools.reduce()` as a synonym for the built-in `reduce()` function.

Python 3.0 adds several new built-in functions and changes the semantics of some existing builtins. Functions that are new in 3.0 such as `bin()` have simply been added to Python 2.6, but existing builtins haven’t been changed; instead, the `future_builtins` module has versions with the new 3.0 semantics. Code written to be compatible with 3.0 can do `from future_builtins import hex, map` as necessary.

A new command-line switch, `-3`, enables warnings about features that will be removed in Python 3.0. You can run code with this switch to see how much work will be necessary to port code to 3.0. The
value of this switch is available to Python code as the boolean variable `sys.py3kwarning`, and to C extension code as `Py_Py3kWarningFlag`.

See also: The 3xxx series of PEPs, which contains proposals for Python 3.0. **PEP 3000** describes the development process for Python 3.0. Start with **PEP 3100** that describes the general goals for Python 3.0, and then explore the higher-numbered PEPs that propose specific features.
Changes to the Development Process

While 2.6 was being developed, the Python development process underwent two significant changes: we switched from SourceForge's issue tracker to a customized Roundup installation, and the documentation was converted from LaTeX to reStructuredText.

New Issue Tracker: Roundup

For a long time, the Python developers had been growing increasingly annoyed by SourceForge's bug tracker. SourceForge's hosted solution doesn't permit much customization; for example, it wasn't possible to customize the life cycle of issues.

The infrastructure committee of the Python Software Foundation therefore posted a call for issue trackers, asking volunteers to set up different products and import some of the bugs and patches from SourceForge. Four different trackers were examined: Jira, Launchpad, Roundup, and Trac. The committee eventually settled on Jira and Roundup as the two candidates. Jira is a commercial product that offers no-cost hosted instances to free-software projects; Roundup is an open-source project that requires volunteers to administer it and a server to host it.

After posting a call for volunteers, a new Roundup installation was set up at [https://bugs.python.org](https://bugs.python.org). One installation of Roundup can host multiple trackers, and this server now also hosts issue trackers for Jython and for the Python web site. It will surely find other uses in the future. Where possible, this edition of “What's New in Python” links to the bug/patch item for each change.

Hosting of the Python bug tracker is kindly provided by Upfront Systems of Stellenbosch, South Africa. Martin von Loewis put a lot of
effort into importing existing bugs and patches from SourceForge; his scripts for this import operation are at http://svn.python.org/view/tracker/importer/ and may be useful to other projects wishing to move from SourceForge to Roundup.

See also:

https://bugs.python.org
   The Python bug tracker.
http://bugs.jython.org:
   The Jython bug tracker.
http://roundup.sourceforge.net/
   Roundup downloads and documentation.
http://svn.python.org/view/tracker/importer/
   Martin von Loewis’s conversion scripts.

New Documentation Format: reStructuredText
Using Sphinx

The Python documentation was written using LaTeX since the project started around 1989. In the 1980s and early 1990s, most documentation was printed out for later study, not viewed online. LaTeX was widely used because it provided attractive printed output while remaining straightforward to write once the basic rules of the markup were learned.

Today LaTeX is still used for writing publications destined for printing, but the landscape for programming tools has shifted. We no longer print out reams of documentation; instead, we browse through it online and HTML has become the most important format to support. Unfortunately, converting LaTeX to HTML is fairly complicated and Fred L. Drake Jr., the long-time Python documentation editor, spent
a lot of time maintaining the conversion process. Occasionally people would suggest converting the documentation into SGML and later XML, but performing a good conversion is a major task and no one ever committed the time required to finish the job.

During the 2.6 development cycle, Georg Brandl put a lot of effort into building a new toolchain for processing the documentation. The resulting package is called Sphinx, and is available from http://sphinx-doc.org/.

Sphinx concentrates on HTML output, producing attractively styled and modern HTML; printed output is still supported through conversion to LaTeX. The input format is reStructuredText, a markup syntax supporting custom extensions and directives that is commonly used in the Python community.

Sphinx is a standalone package that can be used for writing, and almost two dozen other projects (listed on the Sphinx web site) have adopted Sphinx as their documentation tool.

See also:

**Documenting Python**
Describes how to write for Python’s documentation.

**Sphinx**
Documentation and code for the Sphinx toolchain.

**Docutils**
The underlying reStructuredText parser and toolset.
PEP 343: The ‘with’ statement

The previous version, Python 2.5, added the ‘with’ statement as an optional feature, to be enabled by a `from __future__ import with_statement` directive. In 2.6 the statement no longer needs to be specially enabled; this means that `with` is now always a keyword. The rest of this section is a copy of the corresponding section from the “What’s New in Python 2.5” document; if you’re familiar with the ‘with’ statement from Python 2.5, you can skip this section.

The ‘with’ statement clarifies code that previously would use `try...finally` blocks to ensure that clean-up code is executed. In this section, I’ll discuss the statement as it will commonly be used. In the next section, I’ll examine the implementation details and show how to write objects for use with this statement.

The ‘with’ statement is a control-flow structure whose basic structure is:

```
with expression [as variable]:
    with-block
```

The expression is evaluated, and it should result in an object that supports the context management protocol (that is, has `__enter__()` and `__exit__()` methods).

The object's `__enter__()` is called before `with-block` is executed and therefore can run set-up code. It also may return a value that is bound to the name `variable`, if given. (Note carefully that `variable` is not assigned the result of `expression`.)

After execution of the `with-block` is finished, the object's `__exit__()` method is called, even if the block raised an exception, and can
therefore run clean-up code.

Some standard Python objects now support the context management protocol and can be used with the 'with' statement. File objects are one example:

```python
with open('/etc/passwd', 'r') as f:
    for line in f:
        print line
        ... more processing code ...
```

After this statement has executed, the file object in f will have been automatically closed, even if the for loop raised an exception part-way through the block.

**Note:** In this case, f is the same object created by `open()`, because `file.__enter__()` returns `self`.

The `threading` module's locks and condition variables also support the 'with' statement:

```python
lock = threading.Lock()
with lock:
    # Critical section of code
    ...
```

The lock is acquired before the block is executed and always released once the block is complete.

The `localcontext()` function in the `decimal` module makes it easy to save and restore the current decimal context, which encapsulates the desired precision and rounding characteristics for computations:

```python
from decimal import Decimal, Context, localcontext

# Displays with default precision of 28 digits
v = Decimal('578')
```
```python
print v.sqrt()

with localcontext(Context(prec=16)):
    # All code in this block uses a precision of 16 digits.
    # The original context is restored on exiting the block.
    print v.sqrt()
```

Writing Context Managers

Under the hood, the `with` statement is fairly complicated. Most people will only use `with` in company with existing objects and don’t need to know these details, so you can skip the rest of this section if you like. Authors of new objects will need to understand the details of the underlying implementation and should keep reading.

A high-level explanation of the context management protocol is:

- The expression is evaluated and should result in an object called a “context manager”. The context manager must have `__enter__()` and `__exit__()` methods.
- The context manager’s `__enter__()` method is called. The value returned is assigned to `VAR`. If no `as VAR` clause is present, the value is simply discarded.
- The code in `BLOCK` is executed.
- If `BLOCK` raises an exception, the context manager’s `__exit__()` method is called with three arguments, the exception details (type, value, traceback, the same values returned by `sys.exc_info()`, which can also be `None` if no exception occurred). The method’s return value controls whether an exception is re-raised: any false value re-raises the exception, and `True` will result in suppressing it. You’ll only rarely want to suppress the exception, because if you do the author of the code containing the `with` statement will never realize anything went wrong.
- If `BLOCK` didn’t raise an exception, the `__exit__()` method is
Let’s think through an example. I won’t present detailed code but will only sketch the methods necessary for a database that supports transactions.

(For people unfamiliar with database terminology: a set of changes to the database are grouped into a transaction. Transactions can be either committed, meaning that all the changes are written into the database, or rolled back, meaning that the changes are all discarded and the database is unchanged. See any database textbook for more information.)

Let’s assume there’s an object representing a database connection. Our goal will be to let the user write code like this:

```python
db_connection = DatabaseConnection()
with db_connection as cursor:
    cursor.execute('insert into ...')
    cursor.execute('delete from ...')
    # ... more operations ...
```

The transaction should be committed if the code in the block runs flawlessly or rolled back if there’s an exception. Here’s the basic interface for `DatabaseConnection` that I’ll assume:

```python
class DatabaseConnection:
    # Database interface
def cursor(self):
        "Returns a cursor object and starts a new transaction"
def commit(self):
    "Commits current transaction"
def rollback(self):
    "Rolls back current transaction"
```

The `__enter__()` method is pretty easy, having only to start a new transaction. For this application the resulting cursor object would be a useful result, so the method will return it. The user can then add as
cursor to their 'with' statement to bind the cursor to a variable name.

```python
class DatabaseConnection:
    ...
    def __enter__(self):
        # Code to start a new transaction
        cursor = self.cursor()
        return cursor
```

The `__exit__()` method is the most complicated because it's where most of the work has to be done. The method has to check if an exception occurred. If there was no exception, the transaction is committed. The transaction is rolled back if there was an exception.

In the code below, execution will just fall off the end of the function, returning the default value of `None`. `None` is false, so the exception will be re-raised automatically. If you wished, you could be more explicit and add a `return` statement at the marked location.

```python
class DatabaseConnection:
    ...
    def __exit__(self, type, value, tb):
        if tb is None:
            # No exception, so commit
            self.commit()
        else:
            # Exception occurred, so rollback.
            self.rollback()
            # return False
```

The `contextlib` module

The `contextlib` module provides some functions and a decorator that are useful when writing objects for use with the 'with' statement.

The decorator is called `contextmanager()`, and lets you write a single generator function instead of defining a new class. The generator should yield exactly one value. The code up to the `yield` will be
executed as the __enter__() method, and the value yielded will be
the method’s return value that will get bound to the variable in the
‘with’ statement’s as clause, if any. The code after the yield will be
executed in the __exit__() method. Any exception raised in the
block will be raised by the yield statement.

Using this decorator, our database example from the previous
section could be written as:

```python
from contextlib import contextmanager

def db_transaction(connection):
    cursor = connection.cursor()
    try:
        yield cursor
    except:
        connection.rollback()
        raise
    else:
        connection.commit()

db = DatabaseConnection()
with db_transaction(db) as cursor:
    ...
```

The contextlib module also has a nested(mgr1, mgr2, ...) function
that combines a number of context managers so you don’t need to
write nested ‘with’ statements. In this example, the single ‘with’
statement both starts a database transaction and acquires a thread
lock:

```python
lock = threading.Lock()
with nested (db_transaction(db), lock) as (cursor, locked):
    ...
```

Finally, the closing() function returns its argument so that it can be
bound to a variable, and calls the argument’s close() method at the
end of the block.
import urllib, sys
from contextlib import closing

with closing(urllib.urlopen('http://www.yahoo.com')) as f:
    for line in f:
        sys.stdout.write(line)

See also:

PEP 343 - The “with” statement
   PEP written by Guido van Rossum and Nick Coghlan; implemented by Mike Bland, Guido van Rossum, and Neal Norwitz. The PEP shows the code generated for a `with` statement, which can be helpful in learning how the statement works.

The documentation for the `contextlib` module.
PEP 366: Explicit Relative Imports From a Main Module

Python’s -m switch allows running a module as a script. When you ran a module that was located inside a package, relative imports didn’t work correctly.

The fix for Python 2.6 adds a __package__ attribute to modules. When this attribute is present, relative imports will be relative to the value of this attribute instead of the __name__ attribute.

PEP 302-style importers can then set __package__ as necessary. The runpy module that implements the -m switch now does this, so relative imports will now work correctly in scripts running from inside a package.
When you run Python, the module search path `sys.path` usually includes a directory whose path ends in "site-packages". This directory is intended to hold locally-installed packages available to all users using a machine or a particular site installation.

Python 2.6 introduces a convention for user-specific site directories. The directory varies depending on the platform:

- Unix and Mac OS X: `~/.local/`
- Windows: `%APPDATA%/Python`

Within this directory, there will be version-specific subdirectories, such as `lib/python2.6/site-packages` on Unix/Mac OS and `Python26/site-packages` on Windows.

If you don’t like the default directory, it can be overridden by an environment variable. `PYTHONUSERBASE` sets the root directory used for all Python versions supporting this feature. On Windows, the directory for application-specific data can be changed by setting the `APPDATA` environment variable. You can also modify the `site.py` file for your Python installation.

The feature can be disabled entirely by running Python with the `-s` option or setting the `PYTHONNOUSERSITE` environment variable.

See also:

**PEP 370 - Per-user site-packages Directory**

PEP written and implemented by Christian Heimes.
PEP 371: The multiprocessing Package

The new `multiprocessing` package lets Python programs create new processes that will perform a computation and return a result to the parent. The parent and child processes can communicate using queues and pipes, synchronize their operations using locks and semaphores, and can share simple arrays of data.

The `multiprocessing` module started out as an exact emulation of the `threading` module using processes instead of threads. That goal was discarded along the path to Python 2.6, but the general approach of the module is still similar. The fundamental class is the `Process`, which is passed a callable object and a collection of arguments. The `start()` method sets the callable running in a subprocess, after which you can call the `is_alive()` method to check whether the subprocess is still running and the `join()` method to wait for the process to exit.

Here's a simple example where the subprocess will calculate a factorial. The function doing the calculation is written strangely so that it takes significantly longer when the input argument is a multiple of 4.

```python
import time
from multiprocessing import Process, Queue

def factorial(queue, N):
    "Compute a factorial."
    # If N is a multiple of 4, this function will take much longer.
    if (N % 4) == 0:
        time.sleep(.05 * N/4)

    # Calculate the result
    fact = 1L
    for i in range(1, N+1):
        fact *= i
    queue.put(fact)
```

A `Queue` is used to communicate the result of the factorial. The `Queue` object is stored in a global variable. The child process will use the value of the variable when the child was created; because it’s a `Queue`, parent and child can use the object to communicate. (If the parent were to change the value of the global variable, the child’s value would be unaffected, and vice versa.)

Two other classes, `Pool` and `Manager`, provide higher-level interfaces. `Pool` will create a fixed number of worker processes, and requests can then be distributed to the workers by calling `apply()` or `apply_async()` to add a single request, and `map()` or `map_async()` to add a number of requests. The following code uses a `Pool` to spread requests across 5 worker processes and retrieve a list of results:

```python
from multiprocessing import Pool

def factorial(N, dictionary):
    "Compute a factorial."
    ...

p = Pool(5)
result = p.map(factorial, range(1, 1000, 10))
for v in result:
    print v
```
This produces the following output:

```
1
39916800
51090942171709440000
822283865417792281772556288000000
33452526613163807108170062053440751665152000000000
...
```

The other high-level interface, the `Manager` class, creates a separate server process that can hold master copies of Python data structures. Other processes can then access and modify these data structures using proxy objects. The following example creates a shared dictionary by calling the `dict()` method; the worker processes then insert values into the dictionary. (Locking is not done for you automatically, which doesn’t matter in this example. `Manager`'s methods also include `Lock()`, `RLock()`, and `Semaphore()` to create shared locks.)

```python
import time
from multiprocessing import Pool, Manager

def factorial(N, dictionary):
    """Compute a factorial."
    # Calculate the result
    fact = 1L
    for i in range(1, N+1):
        fact = fact * i

    # Store result in dictionary
dictionary[N] = fact

if __name__ == '__main__':
    p = Pool(5)
    mgr = Manager()
    d = mgr.dict()        # Create shared dictionary

    # Run tasks using the pool
    for N in range(1, 1000, 10):
        p.apply_async(factorial, (N, d))
```
Mark pool as closed -- no more tasks can be added.

```
p.close()
```

Wait for tasks to exit

```
p.join()
```

Output results

```
for k, v in sorted(d.items()):
    print k, v
```

This will produce the output:

```
1 1
11 39916800
21 5109942171709440000
31 8222838654177922817725562880000000
41 334525266131638071081700620534407516651520000000000
51 15511187532873822802242430164693032110632597200169861120000.
```

See also: The documentation for the `multiprocessing` module.

**PEP 371 - Addition of the multiprocessing package**

PEP written by Jesse Noller and Richard Oudkerk; implemented by Richard Oudkerk and Jesse Noller.
PEP 3101: Advanced String Formatting

In Python 3.0, the % operator is supplemented by a more powerful string formatting method, `format()`. Support for the `str.format()` method has been backported to Python 2.6.

In 2.6, both 8-bit and Unicode strings have a `format()` method that treats the string as a template and takes the arguments to be formatted. The formatting template uses curly brackets `{, }` as special characters:

```python
>>> # Substitute positional argument 0 into the string.
>>> "User ID: {0}".format("root")
'User ID: root'
>>> # Use the named keyword arguments
>>> "User ID: {uid} Last seen: {last_login}".format(  
...     uid="root",  
...     last_login = "5 Mar 2008 07:20")
'User ID: root Last seen: 5 Mar 2008 07:20'
```

Curly brackets can be escaped by doubling them:

```python
>>> "Empty dict: {{}}".format()
"Empty dict: {}
```

Field names can be integers indicating positional arguments, such as `{0}`, `{1}`, etc. or names of keyword arguments. You can also supply compound field names that read attributes or access dictionary keys:

```python
>>> import sys
>>> print 'Platform: {0.platform}\nPython version: {0.version}'
Platform: darwin
[GCC 4.0.1 (Apple Computer, Inc. build 5367)]
>>> import mimetypes
>>> 'Content-type: {0[.mp4]}'.format(mimetypes.types_map)
```

Note that when using dictionary-style notation such as `[.mp4]`, you don't need to put any quotation marks around the string; it will look up the value using `.mp4` as the key. Strings beginning with a number will be converted to an integer. You can't write more complicated expressions inside a format string.

So far we've shown how to specify which field to substitute into the resulting string. The precise formatting used is also controllable by adding a colon followed by a format specifier. For example:

```python
>>> # Field 0: left justify, pad to 15 characters
>>> # Field 1: right justify, pad to 6 characters
>>> fmt = '{0:15} ${1:>6}'
>>> fmt.format('Registration', 35)
'Registration $ 35'
>>> fmt.format('Tutorial', 50)
'Tutorial $ 50'
>>> fmt.format('Banquet', 125)
'Banquet $ 125'
```

Format specifiers can reference other fields through nesting:

```python
>>> fmt = '{0:{1}}'
>>> width = 15
>>> fmt.format('Invoice #1234', width)
'Invoice #1234 '
>>> width = 35
>>> fmt.format('Invoice #1234', width)
'Invoice #1234 '
```

The alignment of a field within the desired width can be specified:

<table>
<thead>
<tr>
<th>Character</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; (default)</td>
<td>Left-align</td>
</tr>
<tr>
<td>&gt;</td>
<td>Right-align</td>
</tr>
<tr>
<td>^</td>
<td>Center</td>
</tr>
</tbody>
</table>
Format specifiers can also include a presentation type, which controls how the value is formatted. For example, floating-point numbers can be formatted as a general number or in exponential notation:

```python
>>> '{0:g}'.format(3.75)
'3.75'
>>> '{0:e}'.format(3.75)
'3.750000e+00'
```

A variety of presentation types are available. Consult the 2.6 documentation for a complete list; here's a sample:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>Binary. Outputs the number in base 2.</td>
</tr>
<tr>
<td>c</td>
<td>Character. Converts the integer to the corresponding Unicode character before printing.</td>
</tr>
<tr>
<td>d</td>
<td>Decimal Integer. Outputs the number in base 10.</td>
</tr>
<tr>
<td>o</td>
<td>Octal format. Outputs the number in base 8.</td>
</tr>
<tr>
<td>x</td>
<td>Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9.</td>
</tr>
<tr>
<td>e</td>
<td>Exponent notation. Prints the number in scientific notation using the letter ‘e’ to indicate the exponent.</td>
</tr>
<tr>
<td>g</td>
<td>General format. This prints the number as a fixed-point number, unless the number is too large, in which case it switches to ‘e’ exponent notation.</td>
</tr>
<tr>
<td>n</td>
<td>Number. This is the same as ‘g’ (for floats) or ‘d’ (for integers), except that it uses the current locale setting to insert the appropriate number separator characters.</td>
</tr>
<tr>
<td>%</td>
<td>Percentage. Multiplies the number by 100 and displays in fixed (‘f’) format, followed by a percent sign.</td>
</tr>
</tbody>
</table>

Classes and types can define a `__format__()` method to control how they're formatted. It receives a single argument, the format specifier:
There's also a `format()` built-in that will format a single value. It calls the type's `__format__() method with the provided specifier:

```python
>>> format(75.6564, '.2f')
'75.66'
```

See also:

**Format String Syntax**
The reference documentation for format fields.

**PEP 3101 - Advanced String Formatting**
PEP written by Talin. Implemented by Eric Smith.
PEP 3105: print As a Function

The `print` statement becomes the `print()` function in Python 3.0. Making `print()` a function makes it possible to replace the function by doing `def print(...)` or importing a new function from somewhere else.

Python 2.6 has a `__future__` import that removes `print` as language syntax, letting you use the functional form instead. For example:

```python
>>> from __future__ import print_function
>>> print('# of entries', len(dictionary), file=sys.stderr)
```

The signature of the new function is:

```python
def print(*args, sep=' ', end='
', file=None)
```

The parameters are:

- `args`: positional arguments whose values will be printed out.
- `sep`: the separator, which will be printed between arguments.
- `end`: the ending text, which will be printed after all of the arguments have been output.
- `file`: the file object to which the output will be sent.

See also:

**PEP 3105 - Make print a function**

PEP written by Georg Brandl.
One error that Python programmers occasionally make is writing the following code:

```python
try:
    ...
except TypeError, ValueError:
    # Wrong!
    ...
```

The author is probably trying to catch both `TypeError` and `ValueError` exceptions, but this code actually does something different: it will catch `TypeError` and bind the resulting exception object to the local name "ValueError". The `ValueError` exception will not be caught at all. The correct code specifies a tuple of exceptions:

```python
try:
    ...
except (TypeError, ValueError):
    ...
```

This error happens because the use of the comma here is ambiguous: does it indicate two different nodes in the parse tree, or a single node that’s a tuple?

Python 3.0 makes this unambiguous by replacing the comma with the word “as”. To catch an exception and store the exception object in the variable `exc`, you must write:

```python
try:
    ...
except TypeError as exc:
    ...
```

Python 3.0 will only support the use of “as”, and therefore interprets the first example as catching two different exceptions. Python 2.6
supports both the comma and “as”, so existing code will continue to work. We therefore suggest using “as” when writing new Python code that will only be executed with 2.6.

See also:

**PEP 3110 - Catching Exceptions in Python 3000**
PEP written and implemented by Collin Winter.
PEP 3112: Byte Literals

Python 3.0 adopts Unicode as the language’s fundamental string type and denotes 8-bit literals differently, either as b'string' or using a bytes constructor. For future compatibility, Python 2.6 adds bytes as a synonym for the str type, and it also supports the b'' notation.

The 2.6 str differs from 3.0’s bytes type in various ways; most notably, the constructor is completely different. In 3.0, bytes([65, 66, 67]) is 3 elements long, containing the bytes representing ABC; in 2.6, bytes([65, 66, 67]) returns the 12-byte string representing the str() of the list.

The primary use of bytes in 2.6 will be to write tests of object type such as isinstance(x, bytes). This will help the 2to3 converter, which can’t tell whether 2.x code intends strings to contain either characters or 8-bit bytes; you can now use either bytes or str to represent your intention exactly, and the resulting code will also be correct in Python 3.0.

There’s also a __future__ import that causes all string literals to become Unicode strings. This means that \u escape sequences can be used to include Unicode characters:

```python
from __future__ import unicode_literals

s = ('\u751f\u3080\u304e\u3000\u751f\u305f\u307e\u3054'
     '\u3081\u3000\u751f\u305f\u307e\u3054')

print len(s)  # 12 Unicode characters
```

At the C level, Python 3.0 will rename the existing 8-bit string type, called PyStringObject in Python 2.x, to PyBytesObject. Python 2.6
uses `#define` to support using the names `PyBytesObject()`, `PyBytes_Check()`, `PyBytes_FromStringAndSize()`, and all the other functions and macros used with strings.

Instances of the `bytes` type are immutable just as strings are. A new `bytearray` type stores a mutable sequence of bytes:

```python
>>> bytearray([65, 66, 67])
bytearray(b'ABC')
>>> b = bytearray(u'\u21ef\u3244', 'utf-8')
>>> b
bytearray(b'\xe2\x87\xaf\xe3\x89\x84')
>>> b[0] = '\xe3'
>>> b
bytearray(b'\xe3\xe2\x87\xaf\xe3\x89\x84')
>>> unicode(str(b), 'utf-8')
u'\u31ef \u3244'
```

Byte arrays support most of the methods of string types, such as `startswith()`/`endswith()`, `find()`/`rfind()`, and some of the methods of lists, such as `append()`, `pop()`, and `reverse()`.

```python
>>> b = bytearray('ABC')
>>> b.append('d')
>>> b.append(ord('e'))
>>> b
bytearray(b'ABCde')
```

There’s also a corresponding C API, with `PyByteArray_FromObject()`, `PyByteArray_FromStringAndSize()`, and various other functions.

**See also:**

**PEP 3112 - Bytes literals in Python 3000**

PEP written by Jason Orendorff; backported to 2.6 by Christian Heimes.
PEP 3116: New I/O Library

Python’s built-in file objects support a number of methods, but file-like objects don’t necessarily support all of them. Objects that imitate files usually support `read()` and `write()`, but they may not support `readline()`, for example. Python 3.0 introduces a layered I/O library in the `io` module that separates buffering and text-handling features from the fundamental read and write operations.

There are three levels of abstract base classes provided by the `io` module:

- **RawIOBase** defines raw I/O operations: `read()`, `readinto()`, `write()`, `seek()`, `tell()`, `truncate()`, and `close()`. Most of the methods of this class will often map to a single system call. There are also `readable()`, `writable()`, and `seekable()` methods for determining what operations a given object will allow.

  Python 3.0 has concrete implementations of this class for files and sockets, but Python 2.6 hasn’t restructured its file and socket objects in this way.

- **BufferedIOBase** is an abstract base class that buffers data in memory to reduce the number of system calls used, making I/O processing more efficient. It supports all of the methods of `RawIOBase`, and adds a `raw` attribute holding the underlying raw object.

  There are five concrete classes implementing this ABC. `BufferedWriter` and `BufferedReader` are for objects that support write-only or read-only usage that have a `seek()` method for random access. `BufferedRandom` objects support read and write
access upon the same underlying stream, and **BufferedRWPair** is for objects such as TTYs that have both read and write operations acting upon unconnected streams of data. The **BytesIO** class supports reading, writing, and seeking over an in-memory buffer.

- **TextIOBase**: Provides functions for reading and writing strings (remember, strings will be Unicode in Python 3.0), and supporting **universal newlines**. **TextIOBase** defines the `readline()` method and supports iteration upon objects.

There are two concrete implementations. **TextIOWrapper** wraps a buffered I/O object, supporting all of the methods for text I/O and adding a **buffer** attribute for access to the underlying object. **StringIO** simply buffers everything in memory without ever writing anything to disk.

(In Python 2.6, **io.StringIO** is implemented in pure Python, so it’s pretty slow. You should therefore stick with the existing **StringIO** module or **cStringIO** for now. At some point Python 3.0’s **io** module will be rewritten into C for speed, and perhaps the C implementation will be backported to the 2.x releases.)

In Python 2.6, the underlying implementations haven’t been restructured to build on top of the **io** module’s classes. The module is being provided to make it easier to write code that’s forward-compatible with 3.0, and to save developers the effort of writing their own implementations of buffering and text I/O.

**See also:**

**PEP 3116 - New I/O**

PEP written by Daniel Stutzbach, Mike Verdone, and Guido van Rossum. Code by Guido van Rossum, Georg Brandl, Walter
Doerwald, Jeremy Hylton, Martin von Loewis, Tony Lownds, and others.
The buffer protocol is a C-level API that lets Python types exchange pointers into their internal representations. A memory-mapped file can be viewed as a buffer of characters, for example, and this lets another module such as _re_ treat memory-mapped files as a string of characters to be searched.

The primary users of the buffer protocol are numeric-processing packages such as NumPy, which expose the internal representation of arrays so that callers can write data directly into an array instead of going through a slower API. This PEP updates the buffer protocol in light of experience from NumPy development, adding a number of new features such as indicating the shape of an array or locking a memory region.

The most important new C API function is `PyObject_GetBuffer(PyObject *obj, Py_buffer *view, int flags)` which takes an object and a set of flags, and fills in the `Py_buffer` structure with information about the object's memory representation. Objects can use this operation to lock memory in place while an external caller could be modifying the contents, so there's a corresponding `PyBuffer_Release(Py_buffer *view)` to indicate that the external caller is done.

The _flags_ argument to `PyObject_GetBuffer()` specifies constraints upon the memory returned. Some examples are:

- `PyBUF_WRITABLE` indicates that the memory must be writable.
- `PyBUF_LOCK` requests a read-only or exclusive lock on the memory.
- `PyBUF_C_CONTIGUOUS` and `PyBUF_F_CONTIGUOUS` requests a C-contiguous (last dimension varies the fastest) or Fortran-
contiguous (first dimension varies the fastest) array layout.

Two new argument codes for `PyArg_ParseTuple()`, `s*` and `z*`, return locked buffer objects for a parameter.

See also:

**PEP 3118 - Revising the buffer protocol**
PEP written by Travis Oliphant and Carl Banks; implemented by Travis Oliphant.
PEP 3119: Abstract Base Classes

Some object-oriented languages such as Java support interfaces, declaring that a class has a given set of methods or supports a given access protocol. Abstract Base Classes (or ABCs) are an equivalent feature for Python. The ABC support consists of an `abc` module containing a metaclass called `ABCMeta`, special handling of this metaclass by the `isinstance()` and `issubclass()` builtins, and a collection of basic ABCs that the Python developers think will be widely useful. Future versions of Python will probably add more ABCs.

Let’s say you have a particular class and wish to know whether it supports dictionary-style access. The phrase “dictionary-style” is vague, however. It probably means that accessing items with `obj[1]` works. Does it imply that setting items with `obj[2] = value` works? Or that the object will have `keys()`, `values()`, and `items()` methods? What about the iterative variants such as `iterkeys()`? `copy()` and `update()`? Iterating over the object with `iter()`?

The Python 2.6 `collections` module includes a number of different ABCs that represent these distinctions. `Iterable` indicates that a class defines `__iter__()`, and `Container` means the class defines a `__contains__()` method and therefore supports `x in y` expressions. The basic dictionary interface of getting items, setting items, and `keys()`, `values()`, and `items()`, is defined by the `MutableMapping` ABC.

You can derive your own classes from a particular ABC to indicate they support that ABC’s interface:

```python
import collections
```
class Storage(collections.MutableMapping):
 ...

Alternatively, you could write the class without deriving from the desired ABC and instead register the class by calling the ABC's register() method:

```python
import collections
class Storage:
 ...
collections.MutableMapping.register(Storage)
```

For classes that you write, deriving from the ABC is probably clearer. The register() method is useful when you've written a new ABC that can describe an existing type or class, or if you want to declare that some third-party class implements an ABC. For example, if you defined a `PrintableType` ABC, it's legal to do:

```python
# Register Python's types
PrintableType.register(int)
PrintableType.register(float)
PrintableType.register(str)
```

Classes should obey the semantics specified by an ABC, but Python can't check this; it's up to the class author to understand the ABC's requirements and to implement the code accordingly.

To check whether an object supports a particular interface, you can now write:

```python
def func(d):
    if not isinstance(d, collections.MutableMapping):
        raise ValueError("Mapping object expected, not %r" % d)
```

Don't feel that you must now begin writing lots of checks as in the
above example. Python has a strong tradition of duck-typing, where explicit type-checking is never done and code simply calls methods on an object, trusting that those methods will be there and raising an exception if they aren’t. Be judicious in checking for ABCs and only do it where it’s absolutely necessary.

You can write your own ABCs by using `abc.ABCMeta` as the metaclass in a class definition:

```python
from abc import ABCMeta, abstractmethod

class Drawable():
    __metaclass__ = ABCMeta

    @abstractmethod
    def draw(self, x, y, scale=1.0):
        pass

    def draw_doubled(self, x, y):
        self.draw(x, y, scale=2.0)

class Square(Drawable):
    def draw(self, x, y, scale):
        ...
```

In the `Drawable` ABC above, the `draw_doubled()` method renders the object at twice its size and can be implemented in terms of other methods described in `Drawable`. Classes implementing this ABC therefore don’t need to provide their own implementation of `draw_doubled()`, though they can do so. An implementation of `draw()` is necessary, though; the ABC can’t provide a useful generic implementation.

You can apply the `@abstractmethod` decorator to methods such as `draw()` that must be implemented; Python will then raise an exception for classes that don’t define the method. Note that the exception is only raised when you actually try to create an instance
of a subclass lacking the method:

```python
>>> class Circle(Drawable):
    ...    pass
    ...
>>> c = Circle()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Circle with abstract methods
draw
```
PEP 3127: Integer Literal Support and Syntax

Python 3.0 changes the syntax for octal (base-8) integer literals, prefixing them with “0o” or “0O” instead of a leading zero, and adds support for binary (base-2) integer literals, signalled by a “0b” or “0B” prefix.

Python 2.6 doesn’t drop support for a leading 0 signalling an octal number, but it does add support for “0o” and “0b”:

```python
>>> 0o21, 2*8 + 1
(17, 17)
>>> 0b101111
47
```

The `oct()` builtin still returns numbers prefixed with a leading zero, and a new `bin()` builtin returns the binary representation for a number:

```python
>>> oct(42)
'052'
>>> future_builtins.oct(42)
'0052'
>>> bin(173)
'0b10101101'
```

The `int()` and `long()` builtns will now accept the “0o” and “0b” prefixes when base-8 or base-2 are requested, or when the `base` argument is zero (signalling that the base used should be determined from the string):

```python
>>> int('0o52', 0)
42
>>> int('1101', 2)
13
```
>>> int('0b1101', 2)
13
>>> int('0b1101', 0)
13

See also:

PEP 3127 - Integer Literal Support and Syntax

PEP written by Patrick Maupin; backported to 2.6 by Eric Smith.
Decorators have been extended from functions to classes. It's now legal to write:

```python
@foo
@bar
class A:
    pass
```

This is equivalent to:

```python
class A:
    pass
A = foo(bar(A))
```

See also:

**PEP 3129 - Class Decorators**
PEP written by Collin Winter.
PEP 3141: A Type Hierarchy for Numbers

Python 3.0 adds several abstract base classes for numeric types inspired by Scheme's numeric tower. These classes were backported to 2.6 as the `numbers` module.

The most general ABC is `Number`. It defines no operations at all, and only exists to allow checking if an object is a number by doing `isinstance(obj, Number)`.

`Complex` is a subclass of `Number`. Complex numbers can undergo the basic operations of addition, subtraction, multiplication, division, and exponentiation, and you can retrieve the real and imaginary parts and obtain a number’s conjugate. Python’s built-in complex type is an implementation of `Complex`.

`Real` further derives from `Complex`, and adds operations that only work on real numbers: `floor()`, `trunc()`, rounding, taking the remainder mod N, floor division, and comparisons.

`Rational` numbers derive from `Real`, have `numerator` and `denominator` properties, and can be converted to floats. Python 2.6 adds a simple rational-number class, `Fraction`, in the `fractions` module. (It's called `Fraction` instead of `Rational` to avoid a name clash with `numbers.Rational`.)

`Integral` numbers derive from `Rational`, and can be shifted left and right with `<<` and `>>`, combined using bitwise operations such as `&` and `|`, and can be used as array indexes and slice boundaries.

In Python 3.0, the PEP slightly redefines the existing builtins `round()`, `math.floor()`, `math.ceil()`, and adds a new one,
math.trunc(), that's been backported to Python 2.6. math.trunc() rounds toward zero, returning the closest Integral that's between the function's argument and zero.

See also:

PEP 3141 - A Type Hierarchy for Numbers
PEP written by Jeffrey Yasskin.

Scheme's numerical tower, from the Guile manual.

Scheme's number datatypes from the R5RS Scheme specification.

The fractions Module

To fill out the hierarchy of numeric types, the fractions module provides a rational-number class. Rational numbers store their values as a numerator and denominator forming a fraction, and can exactly represent numbers such as 2/3 that floating-point numbers can only approximate.

The Fraction constructor takes two Integral values that will be the numerator and denominator of the resulting fraction.

```python
>>> from fractions import Fraction
>>> a = Fraction(2, 3)
>>> b = Fraction(2, 5)
>>> float(a), float(b)
(0.6666666666666666, 0.40000000000000002)
>>> a+b
Fraction(16, 15)
>>> a/b
Fraction(5, 3)
```

For converting floating-point numbers to rationals, the float type now has an as_integer_ratio() method that returns the numerator and
denominator for a fraction that evaluates to the same floating-point value:

```python
>>> (2.5).as_integer_ratio()
(5, 2)
>>> (3.1415).as_integer_ratio()
(7074029114692207L, 2251799813685248L)
>>> (1./3).as_integer_ratio()
(6004799503160661L, 18014398509481984L)
```

Note that values that can only be approximated by floating-point numbers, such as 1./3, are not simplified to the number being approximated; the fraction attempts to match the floating-point value exactly.

The `fractions` module is based upon an implementation by Sjoerd Mullender that was in Python's `demo/classes` directory for a long time. This implementation was significantly updated by Jeffrey Yasskin.
Some smaller changes made to the core Python language are:

- Directories and zip archives containing a `__main__.py` file can now be executed directly by passing their name to the interpreter. The directory or zip archive is automatically inserted as the first entry in `sys.path`. (Suggestion and initial patch by Andy Chu, subsequently revised by Phillip J. Eby and Nick Coghlan; issue 1739468.)

- The `hasattr()` function was catching and ignoring all errors, under the assumption that they meant a `__getattr__()` method was failing somehow and the return value of `hasattr()` would therefore be `False`. This logic shouldn’t be applied to `KeyboardInterrupt` and `SystemExit`, however; Python 2.6 will no longer discard such exceptions when `hasattr()` encounters them. (Fixed by Benjamin Peterson; issue 2196.)

- When calling a function using the `**` syntax to provide keyword arguments, you are no longer required to use a Python dictionary; any mapping will now work:

```python
>>> def f(**kw):
...     print sorted(kw)
...
>>> ud=UserDict.UserDict()
>>> ud['a'] = 1
>>> ud['b'] = 'string'
>>> f(**ud)
['a', 'b']
```

(Contributed by Alexander Belopolsky; issue 1686487.)

It’s also become legal to provide keyword arguments after a
*args argument to a function call.

```python
>>> def f(*args, **kw):
...     print(args, kw)
... >>> f(1, 2, 3, *(4, 5, 6), keyword=13)
(1, 2, 3, 4, 5, 6) {'keyword': 13}
```

Previously this would have been a syntax error. (Contributed by Amaury Forgeot d'Arc; issue 3473.)

- A new builtin, `next(iterator, [default])` returns the next item from the specified iterator. If the `default` argument is supplied, it will be returned if `iterator` has been exhausted; otherwise, the `StopIteration` exception will be raised. (Backported in issue 2719.)

- Tuples now have `index()` and `count()` methods matching the list type's `index()` and `count()` methods:

```python
>>> t = (0, 1, 2, 3, 4, 0, 1, 2)
>>> t.index(3)
3
>>> t.count(0)
2
```

(Contributed by Raymond Hettinger)

- The built-in types now have improved support for extended slicing syntax, accepting various combinations of `(start, stop, step)`. Previously, the support was partial and certain corner cases wouldn’t work. (Implemented by Thomas Wouters.)

- Properties now have three attributes, `getter`, `setter` and `deleter`, that are decorators providing useful shortcuts for adding a getter, setter or deleter function to an existing property. You would use them like this:
Several methods of the built-in set types now accept multiple iterables: `intersection()`, `intersection_update()`, `union()`, `update()`, `difference()` and `difference_update()`.

```python
>>> s=set('1234567890')
>>> s.intersection('abc123', 'cdf246')  # Intersection between all inputs
set(['2'])
>>> s.difference('246', '789')
set(['1', '0', '3', '5'])
```

(Contributed by Raymond Hettinger.)

Many floating-point features were added. The `float()` function will now turn the string `nan` into an IEEE 754 Not A Number value, and `+inf` and `-inf` into positive or negative infinity. This works on any platform with IEEE 754 semantics. (Contributed by Christian Heimes; issue 1635.)
Other functions in the `math` module, `isinf()` and `isnan()`, return true if their floating-point argument is infinite or Not A Number. (issue 1640)

Conversion functions were added to convert floating-point numbers into hexadecimal strings (issue 3008). These functions convert floats to and from a string representation without introducing rounding errors from the conversion between decimal and binary. Floats have a `hex()` method that returns a string representation, and the `float.fromhex()` method converts a string back into a number:

```python
>>> a = 3.75
>>> a.hex()
'0x1.e000000000000p+1'
>>> float.fromhex('0x1.e000000000000p+1')
3.75
>>> b=1./3
>>> b.hex()
'0x1.5555555555555p-2'
```

- A numerical nicety: when creating a complex number from two floats on systems that support signed zeros (-0 and +0), the `complex()` constructor will now preserve the sign of the zero. (Fixed by Mark T. Dickinson; issue 1507.)

- Classes that inherit a `__hash__()` method from a parent class can set `__hash__ = None` to indicate that the class isn’t hashable. This will make `hash(obj)` raise a `TypeError` and the class will not be indicated as implementing the `Hashable` ABC.

You should do this when you’ve defined a `__cmp__()` or `__eq__()` method that compares objects by their value rather than by identity. All objects have a default hash method that uses `id(obj)` as the hash value. There’s no tidy way to remove the `__hash__()` method inherited from a parent class, so assigning
None was implemented as an override. At the C level, extensions can set `tp_hash` to `PyObject_HashNotImplemented()`. (Fixed by Nick Coghlan and Amaury Forgeot d'Arc; issue 2235.)

- The `GeneratorExit` exception now subclasses `BaseException` instead of `Exception`. This means that an exception handler that does `except Exception:` will not inadvertently catch `GeneratorExit`. (Contributed by Chad Austin; issue 1537.)

- Generator objects now have a `gi_code` attribute that refers to the original code object backing the generator. (Contributed by Collin Winter; issue 1473257.)

- The `compile()` built-in function now accepts keyword arguments as well as positional parameters. (Contributed by Thomas Wouters; issue 1444529.)

- The `complex()` constructor now accepts strings containing parenthesized complex numbers, meaning that `complex(repr(cplx))` will now round-trip values. For example, `complex('(3+4j)')` now returns the value (3+4j). (issue 1491866)

- The string `translate()` method now accepts `None` as the translation table parameter, which is treated as the identity transformation. This makes it easier to carry out operations that only delete characters. (Contributed by Bengt Richter and implemented by Raymond Hettinger; issue 1193128.)

- The built-in `dir()` function now checks for a `__dir__()` method on the objects it receives. This method must return a list of strings containing the names of valid attributes for the object, and lets the object control the value that `dir()` produces. Objects that have `__getattr__()` or `__getattr__()`
methods can use this to advertise pseudo-attributes they will honor. (issue 1591665)

- Instance method objects have new attributes for the object and function comprising the method; the new synonym for im_self is __self__, and im_func is also available as __func__. The old names are still supported in Python 2.6, but are gone in 3.0.

- An obscure change: when you use the locals() function inside a class statement, the resulting dictionary no longer returns free variables. (Free variables, in this case, are variables referenced in the class statement that aren't attributes of the class.)

**Optimizations**

- The warnings module has been rewritten in C. This makes it possible to invoke warnings from the parser, and may also make the interpreter's startup faster. (Contributed by Neal Norwitz and Brett Cannon; issue 1631171.)

- Type objects now have a cache of methods that can reduce the work required to find the correct method implementation for a particular class; once cached, the interpreter doesn't need to traverse base classes to figure out the right method to call. The cache is cleared if a base class or the class itself is modified, so the cache should remain correct even in the face of Python's dynamic nature. (Original optimization implemented by Armin Rigo, updated for Python 2.6 by Kevin Jacobs; issue 1700288.)

By default, this change is only applied to types that are included with the Python core. Extension modules may not necessarily be compatible with this cache, so they must explicitly add Py_TPFLAGS_HAVE_VERSION_TAG to the module's tp_flags field to enable the method cache. (To be compatible with the method
cache, the extension module’s code must not directly access and modify the \texttt{tp\_dict} member of any of the types it implements. Most modules don’t do this, but it’s impossible for the Python interpreter to determine that. See issue 1878 for some discussion.

- Function calls that use keyword arguments are significantly faster by doing a quick pointer comparison, usually saving the time of a full string comparison. (Contributed by Raymond Hettinger, after an initial implementation by Antoine Pitrou; issue 1819.)

- All of the functions in the \texttt{struct} module have been rewritten in C, thanks to work at the Need For Speed sprint. (Contributed by Raymond Hettinger.)

- Some of the standard built-in types now set a bit in their type objects. This speeds up checking whether an object is a subclass of one of these types. (Contributed by Neal Norwitz.)

- Unicode strings now use faster code for detecting whitespace and line breaks; this speeds up the \texttt{split()} method by about 25% and \texttt{splitlines()} by 35%. (Contributed by Antoine Pitrou.) Memory usage is reduced by using pymalloc for the Unicode string’s data.

- The \texttt{with} statement now stores the \texttt{\_exit\_()} method on the stack, producing a small speedup. (Implemented by Jeffrey Yasskin.)

- To reduce memory usage, the garbage collector will now clear internal free lists when garbage-collecting the highest generation of objects. This may return memory to the operating system sooner.
Two command-line options have been reserved for use by other Python implementations. The -J switch has been reserved for use by Jython for Jython-specific options, such as switches that are passed to the underlying JVM. -x has been reserved for options specific to a particular implementation of Python such as CPython, Jython, or IronPython. If either option is used with Python 2.6, the interpreter will report that the option isn't currently used.

Python can now be prevented from writing .pyc or .pyo files by supplying the -B switch to the Python interpreter, or by setting the PYTHONDONTWRITEBYTECODE environment variable before running the interpreter. This setting is available to Python programs as the sys.dont_write_bytecode variable, and Python code can change the value to modify the interpreter's behaviour. (Contributed by Neal Norwitz and Georg Brandl.)

The encoding used for standard input, output, and standard error can be specified by setting the PYTHONIOENCODING environment variable before running the interpreter. The value should be a string in the form <encoding> or <encoding>:<errorhandler>. The encoding part specifies the encoding's name, e.g. utf-8 or latin-1; the optional errorhandler part specifies what to do with characters that can't be handled by the encoding, and should be one of “error”, “ignore”, or “replace”. (Contributed by Martin von Loewis.)
New and Improved Modules

As in every release, Python's standard library received a number of enhancements and bug fixes. Here's a partial list of the most notable changes, sorted alphabetically by module name. Consult the Misc/NEWS file in the source tree for a more complete list of changes, or look through the Subversion logs for all the details.

- The asyncore and asynchat modules are being actively maintained again, and a number of patches and bugfixes were applied. (Maintained by Josiah Carlson; see issue 1736190 for one patch.)

- The bsddb module also has a new maintainer, Jesús Cea Avion, and the package is now available as a standalone package. The web page for the package is www.jcea.es/programacion/pybsddb.htm. The plan is to remove the package from the standard library in Python 3.0, because its pace of releases is much more frequent than Python's.

    The bsddb.dbshelve module now uses the highest pickling protocol available, instead of restricting itself to protocol 1. (Contributed by W. Barnes.)

- The cgi module will now read variables from the query string of an HTTP POST request. This makes it possible to use form actions with URLs that include query strings such as “/cgi-bin/add.py?category=1”. (Contributed by Alexandre Fiori and Nubis; issue 1817.)

    The parse_qs() and parse_qsl() functions have been relocated from the cgi module to the urlparse module. The versions still available in the cgi module will trigger
• The `cmath` module underwent extensive revision, contributed by Mark Dickinson and Christian Heimes. Five new functions were added:

  - `polar()` converts a complex number to polar form, returning the modulus and argument of the complex number.
  - `rect()` does the opposite, turning a modulus, argument pair back into the corresponding complex number.
  - `phase()` returns the argument (also called the angle) of a complex number.
  - `isnan()` returns True if either the real or imaginary part of its argument is a NaN.
  - `isinf()` returns True if either the real or imaginary part of its argument is infinite.

The revisions also improved the numerical soundness of the `cmath` module. For all functions, the real and imaginary parts of the results are accurate to within a few units of least precision (ulps) whenever possible. See issue 1381 for the details. The branch cuts for `asinh()`, `atanh()`: and `atan()` have also been corrected.

The tests for the module have been greatly expanded; nearly 2000 new test cases exercise the algebraic functions.

On IEEE 754 platforms, the `cmath` module now handles IEEE 754 special values and floating-point exceptions in a manner consistent with Annex ‘G’ of the C99 standard.

• A new data type in the `collections` module: `namedtuple(typename, fieldnames)` is a factory function that creates subclasses of the standard tuple whose fields are
accessible by name as well as index. For example:

```python
>>> var_type = collections.namedtuple('variable',
    ...
    'id name type size')
>>> # Names are separated by spaces or commas.
>>> # 'id, name, type, size' would also work.
>>> var_type._fields
('id', 'name', 'type', 'size')

>>> var = var_type(1, 'frequency', 'int', 4)
>>> print var[0], var.id  # Equivalent
1 1
>>> print var[2], var.type  # Equivalent
int int
>>> var._asdict()
{'size': 4, 'type': 'int', 'id': 1, 'name': 'frequency'}
>>> v2 = var._replace(name='amplitude')
>>> v2
variable(id=1, name='amplitude', type='int', size=4)
```

Several places in the standard library that returned tuples have been modified to return `namedtuple` instances. For example, the `Decimal.as_tuple()` method now returns a named tuple with `sign`, `digits`, and `exponent` fields.

(Contributed by Raymond Hettinger.)

- Another change to the `collections` module is that the `deque` type now supports an optional `maxlen` parameter; if supplied, the deque’s size will be restricted to no more than `maxlen` items. Adding more items to a full deque causes old items to be discarded.

```python
>>> from collections import deque
>>> dq=deque(maxlen=3)
>>> dq
deque([], maxlen=3)
>>> dq.append(1); dq.append(2); dq.append(3)
>>> dq
deque([1, 2, 3], maxlen=3)
>>> dq.append(4)
```
deque([2, 3, 4], maxlen=3)

(Contributed by Raymond Hettinger.)

- The `Cookie` module's `Morsel` objects now support an `httponly` attribute. In some browsers, cookies with this attribute set cannot be accessed or manipulated by JavaScript code. (Contributed by Arvin Schnell; issue 1638033.)

- A new window method in the `curses` module, `chgat()`, changes the display attributes for a certain number of characters on a single line. (Contributed by Fabian Kreutz.)

```
# Boldface text starting at y=0, x=21
# and affecting the rest of the line.
stdscr.chgat(0, 21, curses.A_BOLD)
```

The `Textbox` class in the `curses.textpad` module now supports editing in insert mode as well as overwrite mode. Insert mode is enabled by supplying a true value for the `insert_mode` parameter when creating the `Textbox` instance.

- The `datetime` module’s `strftime()` methods now support a `%f` format code that expands to the number of microseconds in the object, zero-padded on the left to six places. (Contributed by Skip Montanaro; issue 1158.)

- The `decimal` module was updated to version 1.66 of the General Decimal Specification. New features include some methods for some basic mathematical functions such as `exp()` and `log10()`:

```
>>> Decimal(1).exp()
Decimal("2.718281828459045235360287471")
>>> Decimal("2.7182818").ln()
Decimal("0.9999999895305022877376682436")
>>> Decimal(1000).log10()
```
The `as_tuple()` method of `Decimal` objects now returns a named tuple with `sign`, `digits`, and `exponent` fields.

(Implemented by Facundo Batista and Mark Dickinson. Named tuple support added by Raymond Hettinger.)

- The `difflib` module’s `SequenceMatcher` class now returns named tuples representing matches, with `a`, `b`, and `size` attributes. (Contributed by Raymond Hettinger.)

- An optional `timeout` parameter, specifying a timeout measured in seconds, was added to the `ftpclient.FTP` class constructor as well as the `connect()` method. (Added by Facundo Batista.) Also, the `FTP` class’s `storbinary()` and `storlines()` now take an optional `callback` parameter that will be called with each block of data after the data has been sent. (Contributed by Phil Schwartz; issue 1221598.)

- The `reduce()` built-in function is also available in the `functools` module. In Python 3.0, the built-in has been dropped and `reduce()` is only available from `functools`; currently there are no plans to drop the built-in in the 2.x series. (Patched by Christian Heimes; issue 1739906.)

- When possible, the `getpass` module will now use `/dev/tty` to print a prompt message and read the password, falling back to standard error and standard input. If the password may be echoed to the terminal, a warning is printed before the prompt is displayed. (Contributed by Gregory P. Smith.)

- The `glob.glob()` function can now return Unicode filenames if a Unicode path was used and Unicode filenames are matched
within the directory. (issue 1001604)

- A new function in the `heapq` module, `merge(iter1, iter2, ...)`, takes any number of iterables returning data in sorted order, and returns a new generator that returns the contents of all the iterators, also in sorted order. For example:

  ```python
  >>> list(heapq.merge([1, 3, 5, 9], [2, 8, 16]))
  [1, 2, 3, 5, 8, 9, 16]
  ```

  Another new function, `heappushpop(heap, item)`, pushes `item` onto `heap`, then pops off and returns the smallest item. This is more efficient than making a call to `heappush()` and then `heappop()`.

  `heapq` is now implemented to only use less-than comparison, instead of the less-than-or-equal comparison it previously used. This makes `heapq`'s usage of a type match the `list.sort()` method. (Contributed by Raymond Hettinger.)

- An optional `timeout` parameter, specifying a timeout measured in seconds, was added to the `httplib.HTTPConnection` and `HTTPSConnection` class constructors. (Added by Facundo Batista.)

- Most of the `inspect` module’s functions, such as `getmoduleinfo()` and `getargs()`, now return named tuples. In addition to behaving like tuples, the elements of the return value can also be accessed as attributes. (Contributed by Raymond Hettinger.)

  Some new functions in the module include `isgenerator()`, `isgeneratorfunction()`, and `isabstract()`.
- The **`itertools`** module gained several new functions.

  **`izip_longest(iter1, iter2, ...[, fillvalue])`** makes tuples from each of the elements; if some of the iterables are shorter than others, the missing values are set to `fillvalue`. For example:

  ```
  >>> tuple(itertools.izip_longest([1,2,3], [1,2,3,4,5]))
  ((1, 1), (2, 2), (3, 3), (None, 4), (None, 5))
  ```

  **`product(iter1, iter2, ..., [repeat=N])`** returns the Cartesian product of the supplied iterables, a set of tuples containing every possible combination of the elements returned from each iterable.

  ```
  >>> list(itertools.product([1,2,3], [4,5,6]))
  [(1, 4), (1, 5), (1, 6),
   (2, 4), (2, 5), (2, 6),
   (3, 4), (3, 5), (3, 6)]
  ```

  The optional `repeat` keyword argument is used for taking the product of an iterable or a set of iterables with themselves, repeated `N` times. With a single iterable argument, `N`-tuples are returned:

  ```
  >>> list(itertools.product([1,2], repeat=3))
  [(1, 1, 1), (1, 1, 2), (1, 2, 1), (1, 2, 2),
   (2, 1, 1), (2, 1, 2), (2, 2, 1), (2, 2, 2)]
  ```

  With two iterables, `2N`-tuples are returned.

  ```
  >>> list(itertools.product([1,2], [3,4], repeat=2))
  [(1, 3, 1, 3), (1, 3, 1, 4), (1, 3, 2, 3), (1, 3, 2, 4),
   (1, 4, 1, 3), (1, 4, 1, 4), (1, 4, 2, 3), (1, 4, 2, 4),
   (2, 3, 1, 3), (2, 3, 1, 4), (2, 3, 2, 3), (2, 3, 2, 4),
   (2, 4, 1, 3), (2, 4, 1, 4), (2, 4, 2, 3), (2, 4, 2, 4)]
  ```

  **`combinations(iterable, r)`** returns sub-sequences of length `r` from the elements of `iterable`. 
```python
>>> list(itertools.combinations('123', 2))
[('1', '2'), ('1', '3'), ('2', '3')]

>>> list(itertools.combinations('123', 3))
[('1', '2', '3')]

>>> list(itertools.combinations('1234', 3))
[('1', '2', '3'), ('1', '2', '4'),
 ('1', '3', '4'), ('2', '3', '4')]
```

`permutations(iter[, r])` returns all the permutations of length `r` of the iterable's elements. If `r` is not specified, it will default to the number of elements produced by the iterable.

```python
>>> list(itertools.permutations([1,2,3,4], 2))
[(1, 2), (1, 3), (1, 4),
 (2, 1), (2, 3), (2, 4),
 (3, 1), (3, 2), (3, 4),
 (4, 1), (4, 2), (4, 3)]
```

`itertools.chain(*iterables)` is an existing function in `itertools` that gained a new constructor in Python 2.6. `itertools.chain.from_iterable(iterable)` takes a single iterable that should return other iterables. `chain()` will then return all the elements of the first iterable, then all the elements of the second, and so on.

```python
>>> list(itertools.chain.from_iterable([[1,2,3], [4,5,6]]))
[1, 2, 3, 4, 5, 6]
```

(All contributed by Raymond Hettinger.)

- The `logging` module's `FileHandler` class and its subclasses `WatchedFileHandler`, `RotatingFileHandler`, and `TimedRotatingFileHandler` now have an optional `delay` parameter to their constructors. If `delay` is true, opening of the log file is deferred until the first `emit()` call is made. (Contributed by Vinay Sajip.)
TimedRotatingFileHandler also has a \texttt{utc} constructor parameter. If the argument is true, UTC time will be used in determining when midnight occurs and in generating filenames; otherwise local time will be used.

- Several new functions were added to the \texttt{math} module:
  
  - \texttt{isinf()} and \texttt{isnan()} determine whether a given float is a (positive or negative) infinity or a NaN (Not a Number), respectively.
  - \texttt{copysign()} copies the sign bit of an IEEE 754 number, returning the absolute value of \texttt{x} combined with the sign bit of \texttt{y}. For example, \texttt{math.copysign(1, -0.0)} returns -1.0. (Contributed by Christian Heimes.)
  - \texttt{factorial()} computes the factorial of a number. (Contributed by Raymond Hettinger; \texttt{issue 2138}.)
  - \texttt{fsum()} adds up the stream of numbers from an iterable, and is careful to avoid loss of precision through using partial sums. (Contributed by Jean Brouwers, Raymond Hettinger, and Mark Dickinson; \texttt{issue 2819}.)
  - \texttt{acosh()}, \texttt{asinh()} and \texttt{atanh()} compute the inverse hyperbolic functions.
  - \texttt{log1p()} returns the natural logarithm of \texttt{1+x} (base \texttt{e}).
  - \texttt{trunc()} rounds a number toward zero, returning the closest \texttt{Integral} that's between the function's argument and zero.
  
  - Added as part of the backport of PEP 3141's type hierarchy for numbers.

- The \texttt{math} module has been improved to give more consistent behaviour across platforms, especially with respect to handling of floating-point exceptions and IEEE 754 special values.

  Whenever possible, the module follows the recommendations of the C99 standard about 754's special values. For example,
sqrt(-1.) should now give a `ValueError` across almost all platforms, while `sqrt(float('NaN'))` should return a NaN on all IEEE 754 platforms. Where Annex ‘F’ of the C99 standard recommends signaling ‘divide-by-zero’ or ‘invalid’, Python will raise `ValueError`. Where Annex ‘F’ of the C99 standard recommends signaling ‘overflow’, Python will raise `OverflowError`. (See issue 711019 and issue 1640.)

(Contributed by Christian Heimes and Mark Dickinson.)

- `mmap` objects now have a `rfind()` method that searches for a substring beginning at the end of the string and searching backwards. The `find()` method also gained an `end` parameter giving an index at which to stop searching. (Contributed by John Lenton.)

- The `operator` module gained a `methodcaller()` function that takes a name and an optional set of arguments, returning a callable that will call the named function on any arguments passed to it. For example:

  ```python
  >>> # Equivalent to lambda s: s.replace('old', 'new')
  >>> replacer = operator.methodcaller('replace', 'old', 'new')
  >>> replacer('old wine in old bottles')
  'new wine in new bottles'
  ```

  (Contributed by Georg Brandl, after a suggestion by Gregory Petrosyan.)

The `attrgetter()` function now accepts dotted names and performs the corresponding attribute lookups:

```python
>>> inst_name = operator.attrgetter('...__class__.__name__')
>>> inst_name('')
'str'
```
The `os` module now wraps several new system calls. `fchmod(fd, mode)` and `fchown(fd, uid, gid)` change the mode and ownership of an opened file, and `lchmod(path, mode)` changes the mode of a symlink. (Contributed by Georg Brandl and Christian Heimes.)

`chflags()` and `lchflags()` are wrappers for the corresponding system calls (where they're available), changing the flags set on a file. Constants for the flag values are defined in the `stat` module; some possible values include `UF_IMMUTABLE` to signal the file may not be changed and `UF_APPEND` to indicate that data can only be appended to the file. (Contributed by M. Levinson.)

`os.closerange(low, high)` efficiently closes all file descriptors from `low` to `high`, ignoring any errors and not including `high` itself. This function is now used by the `subprocess` module to make starting processes faster. (Contributed by Georg Brandl; issue 1663329.)

The `os.environ` object's `clear()` method will now unset the environment variables using `os.unsetenv()` in addition to clearing the object's keys. (Contributed by Martin Horcicka; issue 1181.)

The `os.walk()` function now has a `followlinks` parameter. If set to True, it will follow symlinks pointing to directories and visit the directory's contents. For backward compatibility, the parameter's default value is false. Note that the function can fall into an
infinite recursion if there's a symlink that points to a parent directory. (issue 1273829)

- In the `os.path` module, the `splitext()` function has been changed to not split on leading period characters. This produces better results when operating on Unix's dot-files. For example, `os.path.splitext('.ipython')` now returns `('.ipython', '')` instead of `('', '.ipython')`. (issue 1115886)

A new function, `os.path.relpath(path, start='.'), returns a relative path from the start path, if it's supplied, or from the current working directory to the destination path. (Contributed by Richard Barran; issue 1339796.)

On Windows, `os.path.expandvars()` will now expand environment variables given in the form “%var%”, and “~user” will be expanded into the user’s home directory path. (Contributed by Josiah Carlson; issue 957650.)

- The Python debugger provided by the `pdb` module gained a new command: “run” restarts the Python program being debugged and can optionally take new command-line arguments for the program. (Contributed by Rocky Bernstein; issue 1393667.)

- The `pdb.post_mortem()` function, used to begin debugging a traceback, will now use the traceback returned by `sys.exc_info()` if no traceback is supplied. (Contributed by Facundo Batista; issue 1106316.)

- The `pickletools` module now has an `optimize()` function that takes a string containing a pickle and removes some unused opcodes, returning a shorter pickle that contains the same data structure. (Contributed by Raymond Hettinger.)
A `get_data()` function was added to the `pkgutil` module that returns the contents of resource files included with an installed Python package. For example:

```python
>>> import pkgutil
>>> print(pkgutil.get_data('test', 'exception_hierarchy.txt'))
```

```
BaseException
   +-- SystemExit
   +-- KeyboardInterrupt
   +-- GeneratorExit
   +-- Exception
      +-- StopIteration
      +-- StandardError
...
```

(Contributed by Paul Moore; issue 2439.)

The `pyexpat` module’s `Parser` objects now allow setting their `buffer_size` attribute to change the size of the buffer used to hold character data. (Contributed by Achim Gaedke; issue 1137.)

The `Queue` module now provides queue variants that retrieve entries in different orders. The `PriorityQueue` class stores queued items in a heap and retrieves them in priority order, and `LifoQueue` retrieves the most recently added entries first, meaning that it behaves like a stack. (Contributed by Raymond Hettinger.)

The `random` module’s `Random` objects can now be pickled on a 32-bit system and unpickled on a 64-bit system, and vice versa. Unfortunately, this change also means that Python 2.6’s `Random` objects can’t be unpickled correctly on earlier versions of Python. (Contributed by Shawn Ligocki; issue 1727780.)

The new `triangular(low, high, mode)` function returns random
numbers following a triangular distribution. The returned values are between low and high, not including high itself, and with mode as the most frequently occurring value in the distribution. (Contributed by Wladmir van der Laan and Raymond Hettinger; issue 1681432.)

- Long regular expression searches carried out by the re module will check for signals being delivered, so time-consuming searches can now be interrupted. (Contributed by Josh Hoyt and Ralf Schmitt; issue 846388.)

The regular expression module is implemented by compiling bytecodes for a tiny regex-specific virtual machine. Untrusted code could create malicious strings of bytecode directly and cause crashes, so Python 2.6 includes a verifier for the regex bytecode. (Contributed by Guido van Rossum from work for Google App Engine; issue 3487.)

- The rlcompleter module's Completer.complete() method will now ignore exceptions triggered while evaluating a name. (Fixed by Lorenz Quack; issue 2250.)

- The sched module's scheduler instances now have a read-only queue attribute that returns the contents of the scheduler's queue, represented as a list of named tuples with the fields (time, priority, action, argument). (Contributed by Raymond Hettinger; issue 1861.)

- The select module now has wrapper functions for the Linux epoll() and BSD kqueue() system calls. modify() method was added to the existing poll objects; pollobj.modify(fd, eventmask) takes a file descriptor or file object and an event mask, modifying the recorded event mask for that file. (Contributed by Christian Heimes; issue 1657.)
The `shutil.copytree()` function now has an optional `ignore` argument that takes a callable object. This callable will receive each directory path and a list of the directory’s contents, and returns a list of names that will be ignored, not copied.

The `shutil` module also provides an `ignore_patterns()` function for use with this new parameter. `ignore_patterns()` takes an arbitrary number of glob-style patterns and returns a callable that will ignore any files and directories that match any of these patterns. The following example copies a directory tree, but skips both `.svn` directories and Emacs backup files, which have names ending with `~`:

```python
shutil.copytree('Doc/library', '/tmp/library',
               ignore=shutil.ignore_patterns('*~', '.svn'))
```

(Contributed by Tarek Ziadé; issue 2663.)

Integrating signal handling with GUI handling event loops like those used by Tkinter or GTK+ has long been a problem; most software ends up polling, waking up every fraction of a second to check if any GUI events have occurred. The `signal` module can now make this more efficient. Calling `signal.set_wakeup_fd(fd)` sets a file descriptor to be used; when a signal is received, a byte is written to that file descriptor. There’s also a C-level function, `PySignal_SetWakeupFd()`, for setting the descriptor.

Event loops will use this by opening a pipe to create two descriptors, one for reading and one for writing. The writable descriptor will be passed to `set_wakeup_fd()`, and the readable descriptor will be added to the list of descriptors monitored by the event loop via `select()` or `poll()`. On receiving a signal, a byte will be written and the main event loop will be woken up,
avoiding the need to poll.

(Contributed by Adam Olsen; issue 1583.)

The `siginterrupt()` function is now available from Python code, and allows changing whether signals can interrupt system calls or not. (Contributed by Ralf Schmitt.)

The `setitimer()` and `getitimer()` functions have also been added (where they're available). `setitimer()` allows setting interval timers that will cause a signal to be delivered to the process after a specified time, measured in wall-clock time, consumed process time, or combined process+system time. (Contributed by Guilherme Polo; issue 2240.)

- The `smtplib` module now supports SMTP over SSL thanks to the addition of the `SMTP_SSL` class. This class supports an interface identical to the existing `SMTP` class. (Contributed by Monty Taylor.) Both class constructors also have an optional `timeout` parameter that specifies a timeout for the initial connection attempt, measured in seconds. (Contributed by Facundo Batista.)

An implementation of the LMTP protocol ([RFC 2033](https://tools.ietf.org/html/rfc2033)) was also added to the module. LMTP is used in place of SMTP when transferring e-mail between agents that don’t manage a mail queue. (LMTP implemented by Leif Hedstrom; issue 957003.)

`SMTP.starttls()` now complies with [RFC 3207](https://tools.ietf.org/html/rfc3207) and forgets any knowledge obtained from the server not obtained from the TLS negotiation itself. (Patch contributed by Bill Fenner; issue 829951.)

protocol designed for use in clustered environments. TIPC addresses are 4- or 5-tuples. (Contributed by Alberto Bertogli; issue 1646.)

A new function, `create_connection()`, takes an address and connects to it using an optional timeout value, returning the connected socket object. This function also looks up the address’s type and connects to it using IPv4 or IPv6 as appropriate. Changing your code to use `create_connection()` instead of `socket(socket.AF_INET, ...)` may be all that’s required to make your code work with IPv6.

- The base classes in the `SocketServer` module now support calling a `handle_timeout()` method after a span of inactivity specified by the server’s `timeout` attribute. (Contributed by Michael Pomraning.) The `serve_forever()` method now takes an optional poll interval measured in seconds, controlling how often the server will check for a shutdown request. (Contributed by Pedro Werneck and Jeffrey Yasskin; issue 742598, issue 1193577.)

- The `sqlite3` module, maintained by Gerhard Haering, has been updated from version 2.3.2 in Python 2.5 to version 2.4.1.

- The `struct` module now supports the C99 `_Bool` type, using the format character `'?'.` (Contributed by David Remahl.)

- The `Popen` objects provided by the `subprocess` module now have `terminate()`, `kill()`, and `send_signal()` methods. On Windows, `send_signal()` only supports the `SIGTERM` signal, and all these methods are aliases for the Win32 API function `TerminateProcess()`. (Contributed by Christian Heimes.)

- A new variable in the `sys` module, `float_info`, is an object
containing information derived from the `float.h` file about the platform's floating-point support. Attributes of this object include `mant_dig` (number of digits in the mantissa), `epsilon` (smallest difference between 1.0 and the next largest value representable), and several others. (Contributed by Christian Heimes; issue 1534.)

Another new variable, `dont_write_bytecode`, controls whether Python writes any `.pyc` or `.pyo` files on importing a module. If this variable is true, the compiled files are not written. The variable is initially set on start-up by supplying the `-B` switch to the Python interpreter, or by setting the `PYTHONDONTWRITEBYTECODE` environment variable before running the interpreter. Python code can subsequently change the value of this variable to control whether bytecode files are written or not. (Contributed by Neal Norwitz and Georg Brandl.)

Information about the command-line arguments supplied to the Python interpreter is available by reading attributes of a named tuple available as `sys.flags`. For example, the `verbose` attribute is true if Python was executed in verbose mode, `debug` is true in debugging mode, etc. These attributes are all read-only. (Contributed by Christian Heimes.)

A new function, `getsizeof()`, takes a Python object and returns the amount of memory used by the object, measured in bytes. Built-in objects return correct results; third-party extensions may not, but can define a `__sizeof__()` method to return the object's size. (Contributed by Robert Schuppenies; issue 2898.)

It's now possible to determine the current profiler and tracer functions by calling `sys.getprofile()` and `sys.gettrace()`. (Contributed by Georg Brandl; issue 1648.)
The `tarfile` module now supports POSIX.1-2001 (pax) tarfiles in addition to the POSIX.1-1988 (ustar) and GNU tar formats that were already supported. The default format is GNU tar; specify the `format` parameter to open a file using a different format:

```python
tar = tarfile.open("output.tar", "w",
    format=tarfile.PAX_FORMAT)
```

The new `encoding` and `errors` parameters specify an encoding and an error handling scheme for character conversions. `'strict'`, `'ignore'`, and `'replace'` are the three standard ways Python can handle errors; `'utf-8'` is a special value that replaces bad characters with their UTF-8 representation. (Character conversions occur because the PAX format supports Unicode filenames, defaulting to UTF-8 encoding.)

The `TarFile.add()` method now accepts an `exclude` argument that’s a function that can be used to exclude certain filenames from an archive. The function must take a filename and return true if the file should be excluded or false if it should be archived. The function is applied to both the name initially passed to `add()` and to the names of files in recursively-added directories.

(All changes contributed by Lars Gustäbel).

- An optional `timeout` parameter was added to the `telnetlib.Telnet` class constructor, specifying a timeout measured in seconds. (Added by Facundo Batista.)

- The `tempfile.NamedTemporaryFile` class usually deletes the temporary file it created when the file is closed. This behaviour can now be changed by passing `delete=False` to the
constructor. (Contributed by Damien Miller; issue 1537850.)

A new class, `SpooledTemporaryFile`, behaves like a temporary file but stores its data in memory until a maximum size is exceeded. On reaching that limit, the contents will be written to an on-disk temporary file. (Contributed by Dustin J. Mitchell.)

The `NamedTemporaryFile` and `SpooledTemporaryFile` classes both work as context managers, so you can write

```python
with tempfile.NamedTemporaryFile() as tmp:
    ...
```

(Contributed by Alexander Belopolsky; issue 2021.)

- The `test.test_support` module gained a number of context managers useful for writing tests. `EnvironmentVarGuard()` is a context manager that temporarily changes environment variables and automatically restores them to their old values.

Another context manager, `TransientResource`, can surround calls to resources that may or may not be available; it will catch and ignore a specified list of exceptions. For example, a network test may ignore certain failures when connecting to an external web site:

```python
with test_support.TransientResource(IOError, errno=errno.ETIMEDOUT):
    f = urllib.urlopen('https://sf.net')
    ...
```

Finally, `check_warnings()` resets the `warning` module's warning filters and returns an object that will record all warning messages triggered (issue 3781):

```python
with test_support.check_warnings() as wrec:
    warnings.simplefilter("always")
    # ... code that triggers a warning ...
    assert str(wrec.message) == "function is outdated"
    assert len(wrec.warnings) == 1, "Multiple warnings rais"
(Contributed by Brett Cannon.)

- The `textwrap` module can now preserve existing whitespace at the beginnings and ends of the newly-created lines by specifying `drop_whitespace=False` as an argument:

```python
>>> S = """This sentence has a bunch of ... extra whitespace.""
>>> print textwrap.fill(S, width=15)
This sentence has a bunch of extra whitespace.
```

```
>>> print textwrap.fill(S, drop_whitespace=False, width=15)
This sentence has a bunch of extra whitespace.
```

(Contributed by Dwayne Bailey; issue 1581073.)

- The `threading` module API is being changed to use properties such as `daemon` instead of `setDaemon()` and `isDaemon()` methods, and some methods have been renamed to use underscores instead of camel-case; for example, the `activeCount()` method is renamed to `active_count()`. Both the 2.6 and 3.0 versions of the module support the same properties and renamed methods, but don't remove the old methods. No date has been set for the deprecation of the old APIs in Python 3.x; the old APIs won't be removed in any 2.x version. (Carried out by several people, most notably Benjamin Peterson.)

The `threading` module’s `Thread` objects gained an `ident` property that returns the thread’s identifier, a nonzero integer.
The `timeit` module now accepts callables as well as strings for the statement being timed and for the setup code. Two convenience functions were added for creating `Timer` instances: `repeat(stmt, setup, time, repeat, number)` and `timeit(stmt, setup, time, number)` create an instance and call the corresponding method. (Contributed by Erik Demaine; issue 1533909.)

The `Tkinter` module now accepts lists and tuples for options, separating the elements by spaces before passing the resulting value to Tcl/Tk. (Contributed by Guilherme Polo; issue 2906.)

The `turtle` module for turtle graphics was greatly enhanced by Gregor Lingl. New features in the module include:

- Better animation of turtle movement and rotation.
- Control over turtle movement using the new `delay()`, `tracer()`, and `speed()` methods.
- The ability to set new shapes for the turtle, and to define a new coordinate system.
- Turtles now have an `undo()` method that can roll back actions.
- Simple support for reacting to input events such as mouse and keyboard activity, making it possible to write simple games.
- A `turtle.cfg` file can be used to customize the starting appearance of the turtle's screen.
- The module's docstrings can be replaced by new docstrings that have been translated into another language.

(issue 1513695)
• An optional `timeout` parameter was added to the `urllib.urlopen()` function and the `urllib.ftpwrapper` class constructor, as well as the `urllib2.urlopen()` function. The parameter specifies a timeout measured in seconds. For example:

```python
>>> u = urllib2.urlopen("http://slow.example.com", timeout=3)
Traceback (most recent call last):
  ...  
urllib2.URLError: <urlopen error timed out>

(Added by Facundo Batista.)

• The Unicode database provided by the `unicodedata` module has been updated to version 5.1.0. (Updated by Martin von Loewis; issue 3811.)

• The `warnings` module’s `formatwarning()` and `showwarning()` gained an optional `line` argument that can be used to supply the line of source code. (Added as part of issue 1631171, which re-implemented part of the `warnings` module in C code.)

A new function, `catch_warnings()`, is a context manager intended for testing purposes that lets you temporarily modify the warning filters and then restore their original values (issue 3781).

• The XML-RPC `SimpleXMLRPCServer` and `DocXMLRPCServer` classes can now be prevented from immediately opening and binding to their socket by passing True as the `bind_and_activate` constructor parameter. This can be used to modify the instance’s `allow_reuse_address` attribute before calling the `server_bind()` and `server_activate()` methods to
open the socket and begin listening for connections. (Contributed by Peter Parente; issue 1599845.)

SimpleXMLRPCServer also has a _send_traceback_header attribute; if true, the exception and formatted traceback are returned as HTTP headers “X-Exception” and “X-Traceback”. This feature is for debugging purposes only and should not be used on production servers because the tracebacks might reveal passwords or other sensitive information. (Contributed by Alan McIntyre as part of his project for Google’s Summer of Code 2007.)

- The xmlrpclib module no longer automatically converts datetime.date and datetime.time to the xmlrpclib.DateTime type; the conversion semantics were not necessarily correct for all applications. Code using xmlrpclib should convert date and time instances. (issue 1330538) The code can also handle dates before 1900 (contributed by Ralf Schmitt; issue 2014) and 64-bit integers represented by using `<i8>` in XML-RPC responses (contributed by Riku Lindblad; issue 2985).

- The zipfile module’s ZipFile class now has extract() and extractall() methods that will unpack a single file or all the files in the archive to the current directory, or to a specified directory:

```python
z = zipfile.ZipFile('python-251.zip')

# Unpack a single file, writing it relative
# to the /tmp directory.
z.extract('Python/sysmodule.c', '/tmp')

# Unpack all the files in the archive.
z.extractall()
```

(Contributed by Alan McIntyre; issue 467924.)
The `open()`, `read()` and `extract()` methods can now take either a filename or a `ZipInfo` object. This is useful when an archive accidentally contains a duplicated filename. (Contributed by Graham Horler; issue 1775025.)

Finally, `zipfile` now supports using Unicode filenames for archived files. (Contributed by Alexey Borzenkov; issue 1734346.)

## The ast module

The `ast` module provides an Abstract Syntax Tree representation of Python code, and Armin Ronacher contributed a set of helper functions that perform a variety of common tasks. These will be useful for HTML templating packages, code analyzers, and similar tools that process Python code.

The `parse()` function takes an expression and returns an AST. The `dump()` function outputs a representation of a tree, suitable for debugging:

```python
import ast

t = ast.parse(""
    d = {}
    for i in 'abcdefghijklm':
        d[i + i] = ord(i) - ord('a') + 1
    print d
""
)
print ast.dump(t)
```

This outputs a deeply nested tree:

```plaintext
Module(body=[
    Assign(targets=[
        Name(id='d', ctx=Store())
    ], value=Dict(keys=[], values=[]))
    For(target=Name(id='i', ctx=Store())),
])
```
iter=Str(s='abcdefghijklm'), body=[
Assign(targets=[
    Subscript(value=
        Name(id='d', ctx=Load()),
        slice=
            Index(value=
                BinOp(left=Name(id='i', ctx=Load()), op=Add(),
                    right=Name(id='i', ctx=Load()))), ctx=Store())
], value=
    BinOp(left=
        BinOp(left=
            Call(func=
                Name(id='ord', ctx=Load()), args=[
                    Name(id='i', ctx=Load())
                ], keywords=[], starargs=None, kwargs=None),
            op=Sub(), right=Call(func=
                Name(id='ord', ctx=Load()), args=[
                    Str(s='a')
                ], keywords=[], starargs=None, kwargs=None)),
        op=Add(), right=Num(n=1))
], orelse=[])
Print(dest=None, values=[
    Name(id='d', ctx=Load())
], nl=True)
]

The **literal_eval()** method takes a string or an AST representing a literal expression, parses and evaluates it, and returns the resulting value. A literal expression is a Python expression containing only strings, numbers, dictionaries, etc. but no statements or function calls. If you need to evaluate an expression but cannot accept the security risk of using an **eval()** call, **literal_eval()** will handle it safely:

```python
>>> literal = 'a, b, {2:4, 3:8, 1:2})
>>> print ast.literal_eval(literal)
('a', 'b', {1: 2, 2: 4, 3: 8})
>>> print ast.literal_eval("a + b")
Traceback (most recent call last):
...    ValueError: malformed string
```
The module also includes `NodeVisitor` and `NodeTransformer` classes for traversing and modifying an AST, and functions for common transformations such as changing line numbers.

### The `future_builtins` module

Python 3.0 makes many changes to the repertoire of built-in functions, and most of the changes can’t be introduced in the Python 2.x series because they would break compatibility. The `future_builtins` module provides versions of these built-in functions that can be imported when writing 3.0-compatible code.

The functions in this module currently include:

- **`ascii(obj)`**: equivalent to `repr()`. In Python 3.0, `repr()` will return a Unicode string, while `ascii()` will return a pure ASCII bytestring.
- **`filter(predicate, iterable)`, `map(func, iterable1, ...)`:** the 3.0 versions return iterators, unlike the 2.x builtins which return lists.
- **`hex(value)`, `oct(value)`**: instead of calling the `__hex__()` or `__oct__()` methods, these versions will call the `__index__()` method and convert the result to hexadecimal or octal. `oct()` will use the new `0o` notation for its result.

### The `json` module: JavaScript Object Notation

The new `json` module supports the encoding and decoding of Python types in JSON (Javascript Object Notation). JSON is a lightweight interchange format often used in web applications. For more information about JSON, see [http://www.json.org](http://www.json.org).

`json` comes with support for decoding and encoding most built-in
Python types. The following example encodes and decodes a
dictionary:

```
>>> import json
>>> data = {"spam": "foo", "parrot": 42}
>>> in_json = json.dumps(data) # Encode the data
>>> in_json
'{"parrot": 42, "spam": "foo"}'}
>>> json.loads(in_json) # Decode into a Python object
{"spam": "foo", "parrot": 42}
```

It's also possible to write your own decoders and encoders to
support more types. Pretty-printing of the JSON strings is also
supported.

`json` (originally called simplejson) was written by Bob Ippolito.

## The `plistlib` module: A Property-List Parser

The `.plist` format is commonly used on Mac OS X to store basic
data types (numbers, strings, lists, and dictionaries) by serializing
them into an XML-based format. It resembles the XML-RPC
serialization of data types.

Despite being primarily used on Mac OS X, the format has nothing
Mac-specific about it and the Python implementation works on any
platform that Python supports, so the `plistlib` module has been
promoted to the standard library.

Using the module is simple:

```
import sys
import plistlib
import datetime

# Create data structure
data_struct = dict(lastAccessed=datetime.datetime.now(),
                   version=1,
                   )
```
categories=('Personal','Shared','Private'))

# Create string containing XML.
plist_str = plistlib.writePlistToString(data_struct)
new_struct = plistlib.readPlistFromString(plist_str)
print data_struct
print new_struct

# Write data structure to a file and read it back.
plistlib.writePlist(data_struct, '/tmp/customizations.plist')
new_struct = plistlib.readPlist('/tmp/customizations.plist')

# read/writePlist accepts file-like objects as well as paths.
plistlib.writePlist(data_struct, sys.stdout)

cetypes Enhancements

Thomas Heller continued to maintain and enhance the ctypes module.

cetypes now supports a c_bool datatype that represents the C99 bool type. (Contributed by David Remahl; issue 1649190.)

The ctypes string, buffer and array types have improved support for extended slicing syntax, where various combinations of (start, stop, step) are supplied. (Implemented by Thomas Wouters.)

All ctypes data types now support from_buffer() and from_buffer_copy() methods that create a ctypes instance based on a provided buffer object. from_buffer_copy() copies the contents of the object, while from_buffer() will share the same memory area.

A new calling convention tells ctypes to clear the errno or Win32 LastError variables at the outset of each wrapped call. (Implemented by Thomas Heller; issue 1798.)

You can now retrieve the Unix errno variable after a function call.
When creating a wrapped function, you can supply `use_errno=True` as a keyword parameter to the `DLL()` function and then call the module-level methods `set_errno()` and `get_errno()` to set and retrieve the error value.

The Win32 LastError variable is similarly supported by the `DLL()`, `OleDLL()`, and `winDLL()` functions. You supply `use_last_error=True` as a keyword parameter and then call the module-level methods `set_last_error()` and `get_last_error()`.

The `byref()` function, used to retrieve a pointer to a ctypes instance, now has an optional `offset` parameter that is a byte count that will be added to the returned pointer.

**Improved SSL Support**

Bill Janssen made extensive improvements to Python 2.6’s support for the Secure Sockets Layer by adding a new module, `ssl`, that’s built atop the OpenSSL library. This new module provides more control over the protocol negotiated, the X.509 certificates used, and has better support for writing SSL servers (as opposed to clients) in Python. The existing SSL support in the `socket` module hasn’t been removed and continues to work, though it will be removed in Python 3.0.

To use the new module, you must first create a TCP connection in the usual way and then pass it to the `ssl.wrap_socket()` function. It’s possible to specify whether a certificate is required, and to obtain certificate info by calling the `getpeercert()` method.

**See also:** The documentation for the `ssl` module.
Deprecations and Removals

- String exceptions have been removed. Attempting to use them raises a `TypeError`.

- Changes to the `Exception` interface as dictated by PEP 352 continue to be made. For 2.6, the `message` attribute is being deprecated in favor of the `args` attribute.

- (3.0-warning mode) Python 3.0 will feature a reorganized standard library that will drop many outdated modules and rename others. Python 2.6 running in 3.0-warning mode will warn about these modules when they are imported.

The list of deprecated modules is: `audiodev`, `bgenlocations`, `buildtools`, `bundlebuilder`, `Canvas`, `compiler`, `dircache`, `dl`, `fpformat`, `gensuitemodule`, `ihooks`, `imageop`, `imgfile`, `linuxaudiodev`, `mhlib`, `mimetypes`, `multifile`, `new`, `pure`, `statvfs`, `sunaudiodev`, `test.testall`, and `toaiff`.

- The `gopherlib` module has been removed.

- The `MimeWriter` module and `mimify` module have been deprecated; use the `email` package instead.

- The `md5` module has been deprecated; use the `hashlib` module instead.

- The `posixfile` module has been deprecated; `fcntl.lockf()` provides better locking.

- The `popen2` module has been deprecated; use the `subprocess` module.
• The rgbimg module has been removed.

• The sets module has been deprecated; it’s better to use the built-in set and frozenset types.

• The sha module has been deprecated; use the hashlib module instead.
Build and C API Changes

Changes to Python’s build process and to the C API include:

- Python now must be compiled with C89 compilers (after 19 years!). This means that the Python source tree has dropped its own implementations of `memcpy()` and `strerror()`, which are in the C89 standard library.

- Python 2.6 can be built with Microsoft Visual Studio 2008 (version 9.0), and this is the new default compiler. See the PCbuild directory for the build files. (Implemented by Christian Heimes.)

- On Mac OS X, Python 2.6 can be compiled as a 4-way universal build. The `configure` script can take a `--with-universal-archs=[32-bit|64-bit|all]` switch, controlling whether the binaries are built for 32-bit architectures (x86, PowerPC), 64-bit (x86-64 and PPC-64), or both. (Contributed by Ronald Oussoren.)

- The BerkeleyDB module now has a C API object, available as `bsddb.db.api`. This object can be used by other C extensions that wish to use the `bsddb` module for their own purposes. (Contributed by Duncan Grisby.)

- The new buffer interface, previously described in the PEP 3118 section, adds `PyObject_GetBuffer()` and `PyBuffer_Release()`, as well as a few other functions.

- Python’s use of the C stdio library is now thread-safe, or at least as thread-safe as the underlying library is. A long-standing potential bug occurred if one thread closed a file object while another thread was reading from or writing to the object. In 2.6
file objects have a reference count, manipulated by the
`PyFile_IncUseCount()` and `PyFile_DecUseCount()` functions. File
objects can’t be closed unless the reference count is zero.
`PyFile_IncUseCount()` should be called while the GIL is still held,
before carrying out an I/O operation using the `FILE *` pointer,
and `PyFile_DecUseCount()` should be called immediately after
the GIL is re-acquired. (Contributed by Antoine Pitrou and
Gregory P. Smith.)

- Importing modules simultaneously in two different threads no
  longer deadlocks; it will now raise an `ImportError`. A new API
  function, `PyImport_ImportModuleNoBlock()`, will look for a module
  in `sys.modules` first, then try to import it after acquiring an import
  lock. If the import lock is held by another thread, an `ImportError`
  is raised. (Contributed by Christian Heimes.)

- Several functions return information about the platform’s
  floating-point support. `PyFloat_GetMax()` returns the maximum
  representable floating point value, and `PyFloat_GetMin()` returns
  the minimum positive value. `PyFloat_GetInfo()` returns an
  object containing more information from the `float.h` file, such as
  "mant_dig" (number of digits in the mantissa), "epsilon"
  (smallest difference between 1.0 and the next largest value
  representable), and several others. (Contributed by Christian
  Heimes; issue 1534.)

- C functions and methods that use `PyComplex_AsCComplex()` will
  now accept arguments that have a `__complex__()` method. In
  particular, the functions in the `cmath` module will now accept
  objects with this method. This is a backport of a Python 3.0
  change. (Contributed by Mark Dickinson; issue 1675423.)

- Python’s C API now includes two functions for case-insensitive
string comparisons, PyOS_stricmp(char*, char*) and PyOS_strnicmp(char*, char*, Py_ssize_t). (Contributed by Christian Heimes; issue 1635.)

- Many C extensions define their own little macro for adding integers and strings to the module’s dictionary in the init* function. Python 2.6 finally defines standard macros for adding values to a module, PyModule_AddStringMacro and PyModule_AddIntMacro(). (Contributed by Christian Heimes.)

- Some macros were renamed in both 3.0 and 2.6 to make it clearer that they are macros, not functions. Py_Size() became Py_SIZE(), Py_Type() became Py_TYPE(), and Py_Refcnt() became Py_REFCNT(). The mixed-case macros are still available in Python 2.6 for backward compatibility. (issue 1629)

- Distutils now places C extensions it builds in a different directory when running on a debug version of Python. (Contributed by Collin Winter; issue 1530959.)

- Several basic data types, such as integers and strings, maintain internal free lists of objects that can be re-used. The data structures for these free lists now follow a naming convention: the variable is always named free_list, the counter is always named numfree, and a macro Py<typename>_MAXFREELIST is always defined.

- A new Makefile target, “make patchcheck”, prepares the Python source tree for making a patch: it fixes trailing whitespace in all modified .py files, checks whether the documentation has been changed, and reports whether the Misc/ACKS and Misc/NEWS files have been updated. (Contributed by Brett Cannon.)

Another new target, “make profile-opt”, compiles a Python
binary using GCC’s profile-guided optimization. It compiles Python with profiling enabled, runs the test suite to obtain a set of profiling results, and then compiles using these results for optimization. (Contributed by Gregory P. Smith.)

### Port-Specific Changes: Windows

- The support for Windows 95, 98, ME and NT4 has been dropped. Python 2.6 requires at least Windows 2000 SP4.

- The new default compiler on Windows is Visual Studio 2008 (version 9.0). The build directories for Visual Studio 2003 (version 7.1) and 2005 (version 8.0) were moved into the PC directory. The new `PCbuild` directory supports cross compilation for X64, debug builds and Profile Guided Optimization (PGO). PGO builds are roughly 10% faster than normal builds. (Contributed by Christian Heimes with help from Amaury Forgeot d’Arc and Martin von Loewis.)

- The `msvcrt` module now supports both the normal and wide char variants of the console I/O API. The `getwch()` function reads a keypress and returns a Unicode value, as does the `getwche()` function. The `putwch()` function takes a Unicode character and writes it to the console. (Contributed by Christian Heimes.)

- `os.path.expandvars()` will now expand environment variables in the form “%var%”, and “~user” will be expanded into the user’s home directory path. (Contributed by Josiah Carlson; issue 957650.)

- The `socket` module’s socket objects now have an `ioctl()` method that provides a limited interface to the `WSAIoctl()` system interface.
The \_winreg\_ module now has a function, ExpandEnvironmentStrings(), that expands environment variable references such as %NAME% in an input string. The handle objects provided by this module now support the context protocol, so they can be used in with statements. (Contributed by Christian Heimes.)

\_winreg\_ also has better support for x64 systems, exposing the DisableReflectionKey(), EnableReflectionKey(), and QueryReflectionKey() functions, which enable and disable registry reflection for 32-bit processes running on 64-bit systems. (issue 1753245)

The msilib module’s Record object gained GetInteger() and GetString() methods that return field values as an integer or a string. (Contributed by Floris Bruynooghe; issue 2125.)

**Port-Specific Changes: Mac OS X**

- When compiling a framework build of Python, you can now specify the framework name to be used by providing the --with-framework-name= option to the configure script.
- The macfs module has been removed. This in turn required the macostools.touched() function to be removed because it depended on the macfs module. (issue 1490190)
- Many other Mac OS modules have been deprecated and will be removed in Python 3.0: \_builtinSuites, aepack, aetools, aetypes, applesingle, applerawmain, applerunner, argvemulator, Audio\_mac, auto\_gil, Carbon, cf\_file, CodeWarrior, ColorPicker, EasyDialogs, Explorer, Finder, FrameWork, findertools, ic, ic\_glue, ic\_open, macerrors, MacOS, macfs, macostools, macresource, MiniAEFrame, Nav, Netscape,
OSATerminology, pimp, PixMapWrapper, StdSuites, SystemEvents, Terminal, and terminalcommand.

Port-Specific Changes: IRIX

A number of old IRIX-specific modules were deprecated and will be removed in Python 3.0: al and AL, cd, cddb, cdplayer, CL and cl, DEVICE, ERRNO, FILE, FL and f1, flp, fm, GET, GLWS, GL and gl, IN, IOCTL, jpeg, panelparser, readcd, SV and sv, torgb, videoreader, and WAIT.
Porting to Python 2.6

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

- Classes that aren't supposed to be hashable should set `__hash__ = None` in their definitions to indicate the fact.

- String exceptions have been removed. Attempting to use them raises a `TypeError`.

- The `__init__()` method of `collections.deque` now clears any existing contents of the deque before adding elements from the iterable. This change makes the behavior match `list.__init__()`.`

- `object.__init__()` previously accepted arbitrary arguments and keyword arguments, ignoring them. In Python 2.6, this is no longer allowed and will result in a `TypeError`. This will affect `__init__()` methods that end up calling the corresponding method on `object` (perhaps through using `super()`). See issue 1683368 for discussion.

- The `Decimal` constructor now accepts leading and trailing whitespace when passed a string. Previously it would raise an `InvalidOperation` exception. On the other hand, the `create_decimal()` method of `Context` objects now explicitly disallows extra whitespace, raising a `ConversionSyntax` exception.

- Due to an implementation accident, if you passed a file path to the built-in `__import__()` function, it would actually import the specified file. This was never intended to work, however, and
the implementation now explicitly checks for this case and raises an `ImportError`.

- C API: the `PyImport_Import()` and `PyImport_ImportModule()` functions now default to absolute imports, not relative imports. This will affect C extensions that import other modules.

- C API: extension data types that shouldn’t be hashable should define their `tp_hash` slot to `PyObject_HashNotImplemented()`.

- The `socket` module exception `socket.error` now inherits from `IOError`. Previously it wasn’t a subclass of `StandardError` but now it is, through `IOError`. (Implemented by Gregory P. Smith; issue 1706815.)

- The `xmlrpclib` module no longer automatically converts `datetime.date` and `datetime.time` to the `xmlrpclib.DateTime` type; the conversion semantics were not necessarily correct for all applications. Code using `xmlrpclib` should convert `date` and `time` instances. (issue 1330538)

- (3.0-warning mode) The `Exception` class now warns when accessed using slicing or index access; having `Exception` behave like a tuple is being phased out.

- (3.0-warning mode) Inequality comparisons between two dictionaries or two objects that don’t implement comparison methods are reported as warnings. `dict1 == dict2` still works, but `dict1 < dict2` is being phased out.

Comparisons between cells, which are an implementation detail of Python’s scoping rules, also cause warnings because such comparisons are forbidden entirely in 3.0.
Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Georg Brandl, Steve Brown, Nick Coghlan, Ralph Corderoy, Jim Jewett, Kent Johnson, Chris Lambacher, Martin Michlmayr, Antoine Pitrou, Brian Warner.
This article explains the new features in Python 2.5. The final release of Python 2.5 is scheduled for August 2006; PEP 356 describes the planned release schedule.

The changes in Python 2.5 are an interesting mix of language and library improvements. The library enhancements will be more important to Python’s user community, I think, because several widely-useful packages were added. New modules include ElementTree for XML processing (xml.etree), the SQLite database module (sqlite), and the ctypes module for calling C functions.

The language changes are of middling significance. Some pleasant new features were added, but most of them aren’t features that you’ll use every day. Conditional expressions were finally added to the language using a novel syntax; see section PEP 308: Conditional Expressions. The new ‘with’ statement will make writing cleanup code easier (section PEP 343: The ‘with’ statement). Values can now be passed into generators (section PEP 342: New Generator Features). Imports are now visible as either absolute or relative (section PEP 328: Absolute and Relative Imports). Some corner cases of exception handling are handled better (section PEP 341: Unified try/except/finally). All these improvements are worthwhile, but they’re improvements to one specific language feature or another; none of them are broad modifications to Python’s semantics.

As well as the language and library additions, other improvements and bugfixes were made throughout the source tree. A search through the SVN change logs finds there were 353 patches applied and 458 bugs fixed between Python 2.4 and 2.5. (Both figures are likely to be underestimates.)
This article doesn’t try to be a complete specification of the new features; instead changes are briefly introduced using helpful examples. For full details, you should always refer to the documentation for Python 2.5 at https://docs.python.org. If you want to understand the complete implementation and design rationale, refer to the PEP for a particular new feature.

Comments, suggestions, and error reports for this document are welcome; please e-mail them to the author or open a bug in the Python bug tracker.
PEP 308: Conditional Expressions

For a long time, people have been requesting a way to write conditional expressions, which are expressions that return value A or value B depending on whether a Boolean value is true or false. A conditional expression lets you write a single assignment statement that has the same effect as the following:

```python
if condition:
    x = true_value
else:
    x = false_value
```

There have been endless tedious discussions of syntax on both python-dev and comp.lang.python. A vote was even held that found the majority of voters wanted conditional expressions in some form, but there was no syntax that was preferred by a clear majority. Candidates included C’s `cond ? true_v : false_v`, `if cond then true_v else false_v`, and 16 other variations.

Guido van Rossum eventually chose a surprising syntax:

```python
x = true_value if condition else false_value
```

Evaluation is still lazy as in existing Boolean expressions, so the order of evaluation jumps around a bit. The `condition` expression in the middle is evaluated first, and the `true_value` expression is evaluated only if the condition was true. Similarly, the `false_value` expression is only evaluated when the condition is false.

This syntax may seem strange and backwards; why does the condition go in the *middle* of the expression, and not in the front as in C’s `c ? x : y`? The decision was checked by applying the new syntax to the modules in the standard library and seeing how the
resulting code read. In many cases where a conditional expression is used, one value seems to be the ‘common case’ and one value is an ‘exceptional case’, used only on rarer occasions when the condition isn’t met. The conditional syntax makes this pattern a bit more obvious:

```
contents = ((doc + ' 
') if doc else '')
```

I read the above statement as meaning “here contents is usually assigned a value of doc+'\n'; sometimes doc is empty, in which special case an empty string is returned.” I doubt I will use conditional expressions very often where there isn’t a clear common and uncommon case.

There was some discussion of whether the language should require surrounding conditional expressions with parentheses. The decision was made to not require parentheses in the Python language’s grammar, but as a matter of style I think you should always use them. Consider these two statements:

```
# First version -- no parens
level = 1 if logging else 0

# Second version -- with parens
level = (1 if logging else 0)
```

In the first version, I think a reader’s eye might group the statement into ‘level = 1’, ‘if logging’, ‘else 0’, and think that the condition decides whether the assignment to level is performed. The second version reads better, in my opinion, because it makes it clear that the assignment is always performed and the choice is being made between two values.

Another reason for including the brackets: a few odd combinations of list comprehensions and lambdas could look like incorrect conditional expressions. See PEP 308 for some examples. If you put
parentheses around your conditional expressions, you won’t run into this case.

See also:

**PEP 308 - Conditional Expressions**
PEP written by Guido van Rossum and Raymond D. Hettinger; implemented by Thomas Wouters.
PEP 309: Partial Function Application

The `functools` module is intended to contain tools for functional-style programming.

One useful tool in this module is the `partial()` function. For programs written in a functional style, you'll sometimes want to construct variants of existing functions that have some of the parameters filled in. Consider a Python function \( f(a, b, c) \); you could create a new function \( g(b, c) \) that was equivalent to \( f(1, b, c) \). This is called “partial function application”.

`partial()` takes the arguments `(function, arg1, arg2, ... kwarg1=value1, kwarg2=value2)`. The resulting object is callable, so you can just call it to invoke `function` with the filled-in arguments.

Here's a small but realistic example:

```python
import functools

def log (message, subsystem):
    "Write the contents of 'message' to the specified subsystem"
    print '%s: %s' % (subsystem, message)
...

server_log = functools.partial(log, subsystem='server')
server_log('Unable to open socket')
```

Here's another example, from a program that uses PyGTK. Here a context-sensitive pop-up menu is being constructed dynamically. The callback provided for the menu option is a partially applied version of the `open_item()` method, where the first argument has been provided.

```python
...
class Application:
    def open_item(self, path):
        ...
    def init(self):
        open_func = functools.partial(self.open_item, item_path)
        popup_menu.append ( ("Open", open_func, 1) )

Another function in the functools module is the update_wrapper(wrapper, wrapped) function that helps you write well-behaved decorators. update_wrapper() copies the name, module, and docstring attribute to a wrapper function so that tracebacks inside the wrapped function are easier to understand. For example, you might write:

def my_decorator(f):
    @functools.wraps(f)
    def wrapper(*args, **kwds):
        print 'Calling decorated function'
        return f(*args, **kwds)
    functools.update_wrapper(wrapper, f)
    return wrapper

wraps() is a decorator that can be used inside your own decorators to copy the wrapped function’s information. An alternate version of the previous example would be:

def my_decorator(f):
    @functools.wraps(f)
    def wrapper(*args, **kwds):
        print 'Calling decorated function'
        return f(*args, **kwds)
    return wrapper

See also:

PEP 309 - Partial Function Application
    PEP proposed and written by Peter Harris; implemented by Hye-Shik Chang and Nick Coghlan, with adaptations by Raymond Hettinger.
PEP 314: Metadata for Python Software Packages v1.1

Some simple dependency support was added to Distutils. The `setup()` function now has `requires`, `provides`, and `obsoletes` keyword parameters. When you build a source distribution using the `sdist` command, the dependency information will be recorded in the `PKG-INFO` file.

Another new keyword parameter is `download_url`, which should be set to a URL for the package's source code. This means it's now possible to look up an entry in the package index, determine the dependencies for a package, and download the required packages.

```python
VERSION = '1.0'
ssetup(name='PyPackage',
    version=VERSION,
    requires=['numarray', 'zlib (>=1.1.4)'],
    obsoletes=['OldPackage'],
    download_url='http://www.example.com/pypackage/dist/pkg-% VERSION),
    )
```

Another new enhancement to the Python package index at https://pypi.python.org is storing source and binary archives for a package. The new `upload` Distutils command will upload a package to the repository.

Before a package can be uploaded, you must be able to build a distribution using the `sdist` Distutils command. Once that works, you can run `python setup.py upload` to add your package to the PyPI archive. Optionally you can GPG-sign the package by supplying the `--sign` and `--identity` options.
Package uploading was implemented by Martin von Löwis and Richard Jones.

See also:

**PEP 314 - Metadata for Python Software Packages v1.1**

PEP proposed and written by A.M. Kuchling, Richard Jones, and Fred Drake; implemented by Richard Jones and Fred Drake.
PEP 328: Absolute and Relative Imports

The simpler part of PEP 328 was implemented in Python 2.4: parentheses could now be used to enclose the names imported from a module using the `from ... import ...` statement, making it easier to import many different names.

The more complicated part has been implemented in Python 2.5: importing a module can be specified to use absolute or package-relative imports. The plan is to move toward making absolute imports the default in future versions of Python.

Let’s say you have a package directory like this:

```
pkg/
pkg/__init__.py
pkg/main.py
pkg/string.py
```

This defines a package named `pkg` containing the `pkg.main` and `pkg.string` submodules.

Consider the code in the `main.py` module. What happens if it executes the statement `import string`? In Python 2.4 and earlier, it will first look in the package’s directory to perform a relative import, finds `pkg/string.py`, imports the contents of that file as the `pkg.string` module, and that module is bound to the name `string` in the `pkg.main` module’s namespace.

That’s fine if `pkg.string` was what you wanted. But what if you wanted Python’s standard `string` module? There’s no clean way to ignore `pkg.string` and look for the standard module; generally you had to look at the contents of `sys.modules`, which is slightly unclean.
Holger Krekel's `py.std` package provides a tidier way to perform imports from the standard library, `import py; py.std.string.join()`, but that package isn't available on all Python installations.

Reading code which relies on relative imports is also less clear, because a reader may be confused about which module, `string` or `pkg.string`, is intended to be used. Python users soon learned not to duplicate the names of standard library modules in the names of their packages' submodules, but you can't protect against having your submodule's name being used for a new module added in a future version of Python.

In Python 2.5, you can switch `import`'s behaviour to absolute imports using a `from __future__ import absolute_import` directive. This absolute-import behaviour will become the default in a future version (probably Python 2.7). Once absolute imports are the default, `import string` will always find the standard library's version. It's suggested that users should begin using absolute imports as much as possible, so it's preferable to begin writing `from pkg import string` in your code.

Relative imports are still possible by adding a leading period to the module name when using the `from ... import` form:

```python
# Import names from pkg.string
from .string import name1, name2
# Import pkg.string
from . import string
```

This imports the `string` module relative to the current package, so in `pkg.main` this will import `name1` and `name2` from `pkg.string`. Additional leading periods perform the relative import starting from the parent of the current package. For example, code in the `A.B.C` module can do:
Leading periods cannot be used with the `import modname` form of the import statement, only the `from ... import` form.

See also:

**PEP 328 - Imports: Multi-Line and Absolute/Relative**
PEP written by Aahz; implemented by Thomas Wouters.

[https://pylib.readthedocs.org/](https://pylib.readthedocs.org/)
The py library by Holger Krekel, which contains the `py.std` package.
PEP 338: Executing Modules as Scripts

The -m switch added in Python 2.4 to execute a module as a script gained a few more abilities. Instead of being implemented in C code inside the Python interpreter, the switch now uses an implementation in a new module, runpy.

The runpy module implements a more sophisticated import mechanism so that it's now possible to run modules in a package such as pychecker.checker. The module also supports alternative import mechanisms such as the zipimport module. This means you can add a .zip archive's path to sys.path and then use the -m switch to execute code from the archive.

See also:

PEP 338 - Executing modules as scripts
    PEP written and implemented by Nick Coghlan.
PEP 341: Unified try/except/finally

Until Python 2.5, the `try` statement came in two flavours. You could use a `finally` block to ensure that code is always executed, or one or more `except` blocks to catch specific exceptions. You couldn’t combine both `except` blocks and a `finally` block, because generating the right bytecode for the combined version was complicated and it wasn’t clear what the semantics of the combined statement should be.

Guido van Rossum spent some time working with Java, which does support the equivalent of combining `except` blocks and a `finally` block, and this clarified what the statement should mean. In Python 2.5, you can now write:

```python
try:
    block-1 ...
except Exception1:
    handler-1 ...
except Exception2:
    handler-2 ...
else:
    else-block
finally:
    final-block
```

The code in `block-1` is executed. If the code raises an exception, the various `except` blocks are tested: if the exception is of class `Exception1`, `handler-1` is executed; otherwise if it’s of class `Exception2`, `handler-2` is executed, and so forth. If no exception is raised, the `else-block` is executed.

No matter what happened previously, the `final-block` is executed once the code block is complete and any raised exceptions handled. Even if there’s an error in an exception handler or the `else-block` and
a new exception is raised, the code in the final-block is still run.

See also:

**PEP 341 - Unifying try-except and try-finally**

PEP written by Georg Brandl; implementation by Thomas Lee.
PEP 342: New Generator Features

Python 2.5 adds a simple way to pass values into a generator. As introduced in Python 2.3, generators only produce output; once a generator’s code was invoked to create an iterator, there was no way to pass any new information into the function when its execution is resumed. Sometimes the ability to pass in some information would be useful. Hackish solutions to this include making the generator’s code look at a global variable and then changing the global variable’s value, or passing in some mutable object that callers then modify.

To refresh your memory of basic generators, here’s a simple example:

```python
def counter (maximum):
    i = 0
    while i < maximum:
        yield i
        i += 1
```

When you call `counter(10)`, the result is an iterator that returns the values from 0 up to 9. On encountering the `yield` statement, the iterator returns the provided value and suspends the function’s execution, preserving the local variables. Execution resumes on the following call to the iterator’s `next()` method, picking up after the `yield` statement.

In Python 2.3, `yield` was a statement; it didn’t return any value. In 2.5, `yield` is now an expression, returning a value that can be assigned to a variable or otherwise operated on:

```python
val = (yield i)
```
I recommend that you always put parentheses around a `yield` expression when you’re doing something with the returned value, as in the above example. The parentheses aren’t always necessary, but it’s easier to always add them instead of having to remember when they’re needed.

(PEP 342) explains the exact rules, which are that a `yield` expression must always be parenthesized except when it occurs at the top-level expression on the right-hand side of an assignment. This means you can write `val = yield i` but have to use parentheses when there’s an operation, as in `val = (yield i) + 12`.

Values are sent into a generator by calling its `send(value)()` method. The generator’s code is then resumed and the `yield` expression returns the specified `value`. If the regular `next()` method is called, the `yield` returns `None`.

Here’s the previous example, modified to allow changing the value of the internal counter.

```python
def counter (maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided, change counter
        if val is not None:
            i = val
        else:
            i += 1
And here’s an example of changing the counter:

```
yield will usually return None, so you should always check for this case. Don’t just use its value in expressions unless you're sure that the send() method will be the only method used to resume your generator function.

In addition to send(), there are two other new methods on generators:

- throw(type, value=None, traceback=None)() is used to raise an exception inside the generator; the exception is raised by the yield expression where the generator’s execution is paused.

- close() raises a new GeneratorExit exception inside the generator to terminate the iteration. On receiving this exception, the generator’s code must either raise GeneratorExit or StopIteration. Catching the GeneratorExit exception and returning a value is illegal and will trigger a RuntimeError; if the function raises some other exception, that exception is propagated to the caller. close() will also be called by Python’s garbage collector when the generator is garbage-collected.

If you need to run cleanup code when a GeneratorExit occurs, I suggest using a try: ... finally: suite instead of catching GeneratorExit.

The cumulative effect of these changes is to turn generators from one-way producers of information into both producers and
consumers.

Generators also become *coroutines*, a more generalized form of subroutines. Subroutines are entered at one point and exited at another point (the top of the function, and a `return` statement), but coroutines can be entered, exited, and resumed at many different points (the `yield` statements). We'll have to figure out patterns for using coroutines effectively in Python.

The addition of the `close()` method has one side effect that isn't obvious. `close()` is called when a generator is garbage-collected, so this means the generator's code gets one last chance to run before the generator is destroyed. This last chance means that `try...finally` statements in generators can now be guaranteed to work; the `finally` clause will now always get a chance to run. The syntactic restriction that you couldn't mix `yield` statements with a `try...finally` suite has therefore been removed. This seems like a minor bit of language trivia, but using generators and `try...finally` is actually necessary in order to implement the `with` statement described by PEP 343. I'll look at this new statement in the following section.

Another even more esoteric effect of this change: previously, the `gi_frame` attribute of a generator was always a frame object. It's now possible for `gi_frame` to be `None` once the generator has been exhausted.

---

**See also:**

**PEP 342 - Coroutines via Enhanced Generators**

PEP written by Guido van Rossum and Phillip J. Eby; implemented by Phillip J. Eby. Includes examples of some fancier uses of generators as coroutines.
Earlier versions of these features were proposed in PEP 288 by Raymond Hettinger and PEP 325 by Samuele Pedroni.

The Wikipedia entry for coroutines.

http://www.sidhe.org/~dan/blog/archives/000178.html
An explanation of coroutines from a Perl point of view, written by Dan Sugalski.
The `with` statement clarifies code that previously would use `try...finally` blocks to ensure that clean-up code is executed. In this section, I’ll discuss the statement as it will commonly be used. In the next section, I’ll examine the implementation details and show how to write objects for use with this statement.

The `with` statement is a new control-flow structure whose basic structure is:

```python
with expression [as variable]:
    with-block
```

The expression is evaluated, and it should result in an object that supports the context management protocol (that is, has `__enter__()` and `__exit__()` methods).

The object’s `__enter__()` is called before `with-block` is executed and therefore can run set-up code. It also may return a value that is bound to the name `variable`, if given. (Note carefully that `variable` is *not* assigned the result of `expression`.)

After execution of the `with-block` is finished, the object’s `__exit__()` method is called, even if the block raised an exception, and can therefore run clean-up code.

To enable the statement in Python 2.5, you need to add the following directive to your module:

```python
from __future__ import with_statement
```

The statement will always be enabled in Python 2.6.
Some standard Python objects now support the context management protocol and can be used with the `with` statement. File objects are one example:

```python
with open('/etc/passwd', 'r') as f:
    for line in f:
        print line
... more processing code ...
```

After this statement has executed, the file object in `f` will have been automatically closed, even if the `for` loop raised an exception part-way through the block.

**Note:** In this case, `f` is the same object created by `open()`, because `file.__enter__()` returns `self`.

The `threading` module's locks and condition variables also support the `with` statement:

```python
lock = threading.Lock()
with lock:
    # Critical section of code
    ...
```

The lock is acquired before the block is executed and always released once the block is complete.

The new `localcontext()` function in the `decimal` module makes it easy to save and restore the current decimal context, which encapsulates the desired precision and rounding characteristics for computations:

```python
from decimal import Decimal, Context, localcontext

# Displays with default precision of 28 digits
v = Decimal('578')
print v.sqrt()
```
```python
with localcontext(Context(prec=16)):
    # All code in this block uses a precision of 16 digits.
    # The original context is restored on exiting the block.
    print v.sqrt()
```

## Writing Context Managers

Under the hood, the `with` statement is fairly complicated. Most people will only use `with` in company with existing objects and don’t need to know these details, so you can skip the rest of this section if you like. Authors of new objects will need to understand the details of the underlying implementation and should keep reading.

A high-level explanation of the context management protocol is:

- The expression is evaluated and should result in an object called a “context manager”. The context manager must have `__enter__()` and `__exit__()` methods.
- The context manager’s `__enter__()` method is called. The value returned is assigned to `VAR`. If no `as VAR` clause is present, the value is simply discarded.
- The code in `BLOCK` is executed.
- If `BLOCK` raises an exception, the `__exit__(type, value, traceback)()` is called with the exception details, the same values returned by `sys.exc_info()`. The method’s return value controls whether the exception is re-raised: any false value re-raises the exception, and `True` will result in suppressing it. You’ll only rarely want to suppress the exception, because if you do the author of the code containing the `with` statement will never realize anything went wrong.
- If `BLOCK` didn’t raise an exception, the `__exit__()` method is still called, but `type`, `value`, and `traceback` are all `None`. 
Let’s think through an example. I won’t present detailed code but will only sketch the methods necessary for a database that supports transactions.

(For people unfamiliar with database terminology: a set of changes to the database are grouped into a transaction. Transactions can be either committed, meaning that all the changes are written into the database, or rolled back, meaning that the changes are all discarded and the database is unchanged. See any database textbook for more information.)

Let’s assume there’s an object representing a database connection. Our goal will be to let the user write code like this:

```python
db_connection = DatabaseConnection()
with db_connection as cursor:
    cursor.execute('insert into ...')
    cursor.execute('delete from ...')
    # ... more operations ...
```

The transaction should be committed if the code in the block runs flawlessly or rolled back if there’s an exception. Here’s the basic interface for `DatabaseConnection` that I’ll assume:

```python
class DatabaseConnection:
    # Database interface
    def cursor (self):
        "Returns a cursor object and starts a new transaction"
    def commit (self):
        "Commits current transaction"
    def rollback (self):
        "Rolls back current transaction"
```

The `__enter__()` method is pretty easy, having only to start a new transaction. For this application the resulting cursor object would be a useful result, so the method will return it. The user can then add `as` to their `with` statement to bind the cursor to a variable name.
class DatabaseConnection:
    ...
    def __enter__(self):
        # Code to start a new transaction
        cursor = self.cursor()
        return cursor

The __exit__() method is the most complicated because it’s where most of the work has to be done. The method has to check if an exception occurred. If there was no exception, the transaction is committed. The transaction is rolled back if there was an exception.

In the code below, execution will just fall off the end of the function, returning the default value of None. None is false, so the exception will be re-raised automatically. If you wished, you could be more explicit and add a return statement at the marked location.

class DatabaseConnection:
    ...
    def __exit__(self, type, value, tb):
        if tb is None:
            # No exception, so commit
            self.commit()
        else:
            # Exception occurred, so rollback.
            self.rollback()
            # return False

The contextlib module

The new contextlib module provides some functions and a decorator that are useful for writing objects for use with the ‘with’ statement.

The decorator is called contextmanager(), and lets you write a single generator function instead of defining a new class. The generator should yield exactly one value. The code up to the yield will be
executed as the __enter__() method, and the value yielded will be the method’s return value that will get bound to the variable in the 'with' statement’s as clause, if any. The code after the yield will be executed in the __exit__() method. Any exception raised in the block will be raised by the yield statement.

Our database example from the previous section could be written using this decorator as:

```python
from contextlib import contextmanager

@contextmanager
def db_transaction(connection):
    cursor = connection.cursor()
    try:
        yield cursor
    except:
        connection.rollback()
        raise
    else:
        connection.commit()

db = DatabaseConnection()
with db_transaction(db) as cursor:
    ...
```

The contextlib module also has a nested(mgr1, mgr2, ...()) function that combines a number of context managers so you don’t need to write nested ‘with’ statements. In this example, the single ‘with’ statement both starts a database transaction and acquires a thread lock:

```python
lock = threading.Lock()
with nested(db_transaction(db), lock) as (cursor, locked):
    ...
```

Finally, the closing(object)() function returns object so that it can be bound to a variable, and calls object.close at the end of the
import urllib, sys
from contextlib import closing

with closing(urllib.urlopen('http://www.yahoo.com')) as f:
    for line in f:
        sys.stdout.write(line)

See also:

PEP 343 - The “with” statement
PEP written by Guido van Rossum and Nick Coghlan; implemented by Mike Bland, Guido van Rossum, and Neal Norwitz. The PEP shows the code generated for a `with` statement, which can be helpful in learning how the statement works.

The documentation for the `contextlib` module.
Exception classes can now be new-style classes, not just classic classes, and the built-in `Exception` class and all the standard built-in exceptions (`NameError`, `ValueError`, etc.) are now new-style classes.

The inheritance hierarchy for exceptions has been rearranged a bit. In 2.5, the inheritance relationships are:

```
BaseException      # New in Python 2.5
 |                  |
 |--- KeyboardInterrupt
 |--- SystemExit
 |--- Exception
      |--- (all other current built-in exceptions)
```

This rearrangement was done because people often want to catch all exceptions that indicate program errors. `KeyboardInterrupt` and `SystemExit` aren’t errors, though, and usually represent an explicit action such as the user hitting `Control-C` or code calling `sys.exit()`.

A bare `except:` will catch all exceptions, so you commonly need to list `KeyboardInterrupt` and `SystemExit` in order to re-raise them. The usual pattern is:

```
try:
    ...
except (KeyboardInterrupt, SystemExit):
    raise
except:
    # Log error...
    # Continue running program...
```

In Python 2.5, you can now write `except Exception` to achieve the same result, catching all the exceptions that usually indicate errors but leaving `KeyboardInterrupt` and `SystemExit` alone. As in previous versions, a bare `except:` still catches all exceptions.
The goal for Python 3.0 is to require any class raised as an exception to derive from `BaseException` or some descendant of `BaseException`, and future releases in the Python 2.x series may begin to enforce this constraint. Therefore, I suggest you begin making all your exception classes derive from `Exception` now. It's been suggested that the bare `except:` form should be removed in Python 3.0, but Guido van Rossum hasn't decided whether to do this or not.

Raising of strings as exceptions, as in the statement `raise "Error occurred"`, is deprecated in Python 2.5 and will trigger a warning. The aim is to be able to remove the string-exception feature in a few releases.

**See also:**

**PEP 352 - Required Superclass for Exceptions**

PEP written by Brett Cannon and Guido van Rossum; implemented by Brett Cannon.
PEP 353: Using ssize_t as the index type

A wide-ranging change to Python’s C API, using a new Py_ssize_t type definition instead of int, will permit the interpreter to handle more data on 64-bit platforms. This change doesn’t affect Python’s capacity on 32-bit platforms.

Various pieces of the Python interpreter used C’s int type to store sizes or counts; for example, the number of items in a list or tuple were stored in an int. The C compilers for most 64-bit platforms still define int as a 32-bit type, so that meant that lists could only hold up to $2^{31} - 1 = 2147483647$ items. (There are actually a few different programming models that 64-bit C compilers can use – see http://www.unix.org/version2/whatsnew/lp64_wp.html for a discussion – but the most commonly available model leaves int as 32 bits.)

A limit of 2147483647 items doesn’t really matter on a 32-bit platform because you’ll run out of memory before hitting the length limit. Each list item requires space for a pointer, which is 4 bytes, plus space for a PyObject representing the item. 2147483647*4 is already more bytes than a 32-bit address space can contain.

It’s possible to address that much memory on a 64-bit platform, however. The pointers for a list that size would only require 16 GiB of space, so it’s not unreasonable that Python programmers might construct lists that large. Therefore, the Python interpreter had to be changed to use some type other than int, and this will be a 64-bit type on 64-bit platforms. The change will cause incompatibilities on 64-bit machines, so it was deemed worth making the transition now, while the number of 64-bit users is still relatively small. (In 5 or 10 years, we may all be on 64-bit machines, and the transition would be
more painful then.)

This change most strongly affects authors of C extension modules. Python strings and container types such as lists and tuples now use `Py_ssize_t` to store their size. Functions such as `PyList_Size()` now return `Py_ssize_t`. Code in extension modules may therefore need to have some variables changed to `Py_ssize_t`.

The `PyArg_ParseTuple()` and `Py_BuildValue()` functions have a new conversion code, `n`, for `Py_ssize_t`. `PyArg_ParseTuple()`’s `s#` and `t#` still output `int` by default, but you can define the macro `PY_SSIZE_T_CLEAN` before including `Python.h` to make them return `Py_ssize_t`.

**PEP 353** has a section on conversion guidelines that extension authors should read to learn about supporting 64-bit platforms.

See also:

**PEP 353 - Using ssize_t as the index type**

PEP written and implemented by Martin von Löwis.
The NumPy developers had a problem that could only be solved by adding a new special method, \_\_index\_(). When using slice notation, as in \[start:stop:step\], the values of the \texttt{start}, \texttt{stop}, and \texttt{step} indexes must all be either integers or long integers. NumPy defines a variety of specialized integer types corresponding to unsigned and signed integers of 8, 16, 32, and 64 bits, but there was no way to signal that these types could be used as slice indexes.

Slicing can’t just use the existing \_\_int\_() method because that method is also used to implement coercion to integers. If slicing used \_\_int\_(), floating-point numbers would also become legal slice indexes and that’s clearly an undesirable behaviour.

Instead, a new special method called \_\_index\_() was added. It takes no arguments and returns an integer giving the slice index to use. For example:

```python
class C:
    def __index__(self):
        return self.value
```

The return value must be either a Python integer or long integer. The interpreter will check that the type returned is correct, and raises a \texttt{TypeError} if this requirement isn’t met.

A corresponding \texttt{nb\_index} slot was added to the C-level \texttt{PyNumberMethods} structure to let C extensions implement this protocol. \texttt{PyNumber\_Index(obj)()} can be used in extension code to call the \_\_index\_() function and retrieve its result.

\textbf{See also:}
PEP 357 - Allowing Any Object to be Used for Slicing

PEP written and implemented by Travis Oliphant.
Here are all of the changes that Python 2.5 makes to the core Python language.

- The `dict` type has a new hook for letting subclasses provide a default value when a key isn’t contained in the dictionary. When a key isn’t found, the dictionary's `__missing__(key)()` method will be called. This hook is used to implement the new `defaultdict` class in the `collections` module. The following example defines a dictionary that returns zero for any missing key:

```python
class zerodict (dict):
    def __missing__ (self, key):
        return 0

d = zerodict({1:1, 2:2})
print d[1], d[2]  # Prints 1, 2
print d[3], d[4]  # Prints 0, 0
```

- Both 8-bit and Unicode strings have new `partition(sep)()` and `rpartition(sep)()` methods that simplify a common use case.

  The `find(S)()` method is often used to get an index which is then used to slice the string and obtain the pieces that are before and after the separator. `partition(sep)()` condenses this pattern into a single method call that returns a 3-tuple containing the substring before the separator, the separator itself, and the substring after the separator. If the separator isn’t found, the first element of the tuple is the entire string and the other two elements are empty. `rpartition(sep)()` also returns a 3-tuple but starts searching from the end of the string; the `r` stands for ‘reverse’.
Some examples:

```python
>>> ('http://www.python.org').partition('://')
('http', '://', 'www.python.org')
>>> ('file:/usr/share/doc/index.html').partition('://')
('file:/usr/share/doc/index.html', '', '')
>>> (u'Subject: a quick question').partition(':')
(u'Subject', u':', u'a quick question')
>>> 'www.python.org'.rpartition('.')
('', '', 'www.python.org')
```

(Implemented by Fredrik Lundh following a suggestion by Raymond Hettinger.)

- The `startswith()` and `endswith()` methods of string types now accept tuples of strings to check for.

```python
def is_image_file(filename):
    return filename.endswith(('gif', 'jpg', 'tiff'))
```

(Implemented by Georg Brandl following a suggestion by Tom Lynn.)

- The `min()` and `max()` built-in functions gained a `key` keyword parameter analogous to the `key` argument for `sort()`. This parameter supplies a function that takes a single argument and is called for every value in the list; `min()/max()` will return the element with the smallest/largest return value from this function. For example, to find the longest string in a list, you can do:

```python
L = ['medium', 'longest', 'short']
# Prints 'longest'
print max(L, key=len)
# Prints 'short', because lexicographically 'short' has the
print max(L)
```
Two new built-in functions, `any()` and `all()`, evaluate whether an iterator contains any true or false values. `any()` returns `True` if any value returned by the iterator is true; otherwise it will return `False`. `all()` returns `True` only if all of the values returned by the iterator evaluate as true. (Suggested by Guido van Rossum, and implemented by Raymond Hettinger.)

The result of a class's `__hash__()` method can now be either a long integer or a regular integer. If a long integer is returned, the hash of that value is taken. In earlier versions the hash value was required to be a regular integer, but in 2.5 the `id()` built-in was changed to always return non-negative numbers, and users often seem to use `id(self)` in `__hash__()` methods (though this is discouraged).

ASCII is now the default encoding for modules. It’s now a syntax error if a module contains string literals with 8-bit characters but doesn’t have an encoding declaration. In Python 2.4 this triggered a warning, not a syntax error. See PEP 263 for how to declare a module’s encoding; for example, you might add a line like this near the top of the source file:

```
# -*- coding: latin1 -*-
```

A new warning, `UnicodeWarning`, is triggered when you attempt to compare a Unicode string and an 8-bit string that can’t be converted to Unicode using the default ASCII encoding. The result of the comparison is false:

```
>>> chr(128) == unichr(128)  # Can't convert chr(128) to Unicode
__main__:1: UnicodeWarning: Unicode equal comparison failed to convert both arguments to Unicode - interpreting them as being unequal
False
```
>>> chr(127) == unichr(127)  # chr(127) can be converted
True

Previously this would raise a `UnicodeDecodeError` exception, but in 2.5 this could result in puzzling problems when accessing a dictionary. If you looked up `unichr(128)` and `chr(128)` was being used as a key, you’d get a `UnicodeDecodeError` exception. Other changes in 2.5 resulted in this exception being raised instead of suppressed by the code in `dictobject.c` that implements dictionaries.

Raising an exception for such a comparison is strictly correct, but the change might have broken code, so instead `UnicodeWarning` was introduced.

(Implemented by Marc-André Lemburg.)

- One error that Python programmers sometimes make is forgetting to include an `__init__.py` module in a package directory. Debugging this mistake can be confusing, and usually requires running Python with the `-v` switch to log all the paths searched. In Python 2.5, a new `ImportWarning` warning is triggered when an import would have picked up a directory as a package but no `__init__.py` was found. This warning is silently ignored by default; provide the `-wd` option when running the Python executable to display the warning message. (Implemented by Thomas Wouters.)

- The list of base classes in a class definition can now be empty. As an example, this is now legal:

```python
class C():
    pass
```
Interactive Interpreter Changes

In the interactive interpreter, `quit` and `exit` have long been strings so that new users get a somewhat helpful message when they try to quit:

```python
>>> quit
'Use Ctrl-D (i.e. EOF) to exit.'
```

In Python 2.5, `quit` and `exit` are now objects that still produce string representations of themselves, but are also callable. Newbies who try `quit()` or `exit()` will now exit the interpreter as they expect. (Implemented by Georg Brandl.)

The Python executable now accepts the standard long options `--help` and `--version`; on Windows, it also accepts the `/?` option for displaying a help message. (Implemented by Georg Brandl.)

Optimizations

Several of the optimizations were developed at the NeedForSpeed sprint, an event held in Reykjavik, Iceland, from May 21–28 2006. The sprint focused on speed enhancements to the CPython implementation and was funded by EWT LLC with local support from CCP Games. Those optimizations added at this sprint are specially marked in the following list.

- When they were introduced in Python 2.4, the built-in `set` and `frozenset` types were built on top of Python’s dictionary type. In 2.5 the internal data structure has been customized for implementing sets, and as a result sets will use a third less memory and are somewhat faster. (Implemented by Raymond
The speed of some Unicode operations, such as finding substrings, string splitting, and character map encoding and decoding, has been improved. (Substring search and splitting improvements were added by Fredrik Lundh and Andrew Dalke at the NeedForSpeed sprint. Character maps were improved by Walter Dörwald and Martin von Löwis.)

The `long(str, base)()` function is now faster on long digit strings because fewer intermediate results are calculated. The peak is for strings of around 800–1000 digits where the function is 6 times faster. (Contributed by Alan McIntyre and committed at the NeedForSpeed sprint.)

It's now illegal to mix iterating over a file with `for line in file` and calling the file object’s `read()/readline()/readlines()` methods. Iteration uses an internal buffer and the `read*()` methods don’t use that buffer. Instead they would return the data following the buffer, causing the data to appear out of order. Mixing iteration and these methods will now trigger a `ValueError` from the `read*()` method. (Implemented by Thomas Wouters.)

The `struct` module now compiles structure format strings into an internal representation and caches this representation, yielding a 20% speedup. (Contributed by Bob Ippolito at the NeedForSpeed sprint.)

The `re` module got a 1 or 2% speedup by switching to Python’s allocator functions instead of the system’s `malloc()` and `free()`. (Contributed by Jack Diederich at the NeedForSpeed sprint.)

The code generator’s peephole optimizer now performs simple constant folding in expressions. If you write something like `a = 2+3`, the code generator will do the arithmetic and produce code corresponding to `a = 5`. (Proposed and implemented by Raymond Hettinger.)

Function calls are now faster because code objects now keep...
the most recently finished frame (a “zombie frame”) in an internal field of the code object, reusing it the next time the code object is invoked. (Original patch by Michael Hudson, modified by Armin Rigo and Richard Jones; committed at the NeedForSpeed sprint.) Frame objects are also slightly smaller, which may improve cache locality and reduce memory usage a bit. (Contributed by Neal Norwitz.)

- Python's built-in exceptions are now new-style classes, a change that speeds up instantiation considerably. Exception handling in Python 2.5 is therefore about 30% faster than in 2.4. (Contributed by Richard Jones, Georg Brandl and Sean Reifschneider at the NeedForSpeed sprint.)

- Importing now caches the paths tried, recording whether they exist or not so that the interpreter makes fewer open() and stat() calls on startup. (Contributed by Martin von Löwis and Georg Brandl.)
New, Improved, and Removed Modules

The standard library received many enhancements and bug fixes in Python 2.5. Here’s a partial list of the most notable changes, sorted alphabetically by module name. Consult the Misc/NEWS file in the source tree for a more complete list of changes, or look through the SVN logs for all the details.

- The `audioop` module now supports the a-LAW encoding, and the code for u-LAW encoding has been improved. (Contributed by Lars Immisch.)

- The `codecs` module gained support for incremental codecs. The `codec.lookup()` function now returns a `CodecInfo` instance instead of a tuple. `CodecInfo` instances behave like a 4-tuple to preserve backward compatibility but also have the attributes `encode`, `decode`, `incrementalencoder`, `incrementaldecoder`, `streamwriter`, and `streamreader`. Incremental codecs can receive input and produce output in multiple chunks; the output is the same as if the entire input was fed to the non-incremental codec. See the `codecs` module documentation for details. (Designed and implemented by Walter Dörwald.)

- The `collections` module gained a new type, `defaultdict`, that subclasses the standard `dict` type. The new type mostly behaves like a dictionary but constructs a default value when a key isn’t present, automatically adding it to the dictionary for the requested key value.

  The first argument to `defaultdict`’s constructor is a factory function that gets called whenever a key is requested but not found. This factory function receives no arguments, so you can
use built-in type constructors such as \texttt{list()} or \texttt{int()}. For example, you can make an index of words based on their initial letter like this:

```python
words = """Nel mezzo del cammin di nostra vita
mi ritrovo per una selva oscura
che la diritta via era smarrita""".lower().split()

index = defaultdict(list)
for w in words:
    init_letter = w[0]
    index[init_letter].append(w)
```

Printing \texttt{index} results in the following output:

```
defaultdict(<type 'list'>, {'c': ['cammin', 'che'], 'e': ['d': ['del', 'di', 'diritta'], 'm': ['mezzo', 'mi']
'l': ['la'], 'o': ['oscura'], 'n': ['nel', 'nostra'
'p': ['per'], 's': ['selva', 'smarrita'],
'r': ['ritrova'], 'u': ['una'], 'v': ['vita', 'via

(Contributed by Guido van Rossum.)
```

- The \texttt{deque} double-ended queue type supplied by the \texttt{collections} module now has a \texttt{remove(value)()} method that removes the first occurrence of \texttt{value} in the queue, raising \texttt{ValueError} if the value isn’t found. (Contributed by Raymond Hettinger.)

- New module: The \texttt{contextlib} module contains helper functions for use with the new \texttt{with} statement. See section \texttt{The contextlib module} for more about this module.

- New module: The \texttt{cProfile} module is a C implementation of the existing \texttt{profile} module that has much lower overhead. The module’s interface is the same as \texttt{profile}: you run
cProfile.run('main()') to profile a function, can save profile data to a file, etc. It’s not yet known if the Hotshot profiler, which is also written in C but doesn’t match the profile module’s interface, will continue to be maintained in future versions of Python. (Contributed by Armin Rigo.)

Also, the pstats module for analyzing the data measured by the profiler now supports directing the output to any file object by supplying a stream argument to the Stats constructor. (Contributed by Skip Montanaro.)

- The csv module, which parses files in comma-separated value format, received several enhancements and a number of bugfixes. You can now set the maximum size in bytes of a field by calling the csv.field_size_limit(new_limit) function; omitting the new_limit argument will return the currently-set limit. The reader class now has a line_num attribute that counts the number of physical lines read from the source; records can span multiple physical lines, so line_num is not the same as the number of records read.

The CSV parser is now stricter about multi-line quoted fields. Previously, if a line ended within a quoted field without a terminating newline character, a newline would be inserted into the returned field. This behavior caused problems when reading files that contained carriage return characters within fields, so the code was changed to return the field without inserting newlines. As a consequence, if newlines embedded within fields are important, the input should be split into lines in a manner that preserves the newline characters.

(Contributed by Skip Montanaro and Andrew McNamara.)

- The datetime class in the datetime module now has a
**strptime(string, format)()** method for parsing date strings, contributed by Josh Spoerri. It uses the same format characters as **time.strptime()** and **time.strftime()**:

```python
from datetime import datetime

ts = datetime.strptime('10:13:15 2006-03-07', '%H:%M:%S %Y-%m-%d')
```

- The **SequenceMatcher.get_matching_blocks()** method in the **difflib** module now guarantees to return a minimal list of blocks describing matching subsequences. Previously, the algorithm would occasionally break a block of matching elements into two list entries. (Enhancement by Tim Peters.)

- The **doctest** module gained a **SKIP** option that keeps an example from being executed at all. This is intended for code snippets that are usage examples intended for the reader and aren’t actually test cases.

  An **encoding** parameter was added to the **testfile()** function and the **DocFileSuite** class to specify the file’s encoding. This makes it easier to use non-ASCII characters in tests contained within a docstring. (Contributed by Bjorn Tillenius.)

- The **email** package has been updated to version 4.0. (Contributed by Barry Warsaw.)

- The **fileinput** module was made more flexible. Unicode filenames are now supported, and a **mode** parameter that defaults to "r" was added to the **input()** function to allow opening files in binary or universal newlines mode. Another new parameter, **openhook**, lets you use a function other than **open()** to open the input files. Once you’re iterating over the set of files, the **FileInput** object’s new **fileno()** returns the file descriptor
for the currently opened file. (Contributed by Georg Brandl.)

- In the \texttt{gc} module, the new \texttt{get_count()} function returns a 3-tuple containing the current collection counts for the three GC generations. This is accounting information for the garbage collector; when these counts reach a specified threshold, a garbage collection sweep will be made. The existing \texttt{gc.collect()} function now takes an optional \texttt{generation} argument of 0, 1, or 2 to specify which generation to collect. (Contributed by Barry Warsaw.)

- The \texttt{nsmallest()} and \texttt{nlargest()} functions in the \texttt{heapq} module now support a \texttt{key} keyword parameter similar to the one provided by the \texttt{min()/max()} functions and the \texttt{sort()} methods. For example:

  ```python
  >>> import heapq
  >>> L = ['short', 'medium', 'longest', 'longer still']
  >>> heapq.nsmallest(2, L)  # Return two lowest elements, lexicographically
  ['longer still', 'longest']
  >>> heapq.nsmallest(2, L, key=len)  # Return two shortest
  ['short', 'medium']
  ```

  (Contributed by Raymond Hettinger.)

- The \texttt{itertools.islice()} function now accepts \texttt{None} for the start and step arguments. This makes it more compatible with the attributes of slice objects, so that you can now write the following:

  ```python
  s = slice(5)  # Create slice object
  itertools.islice(iterable, s.start, s.stop, s.step)
  ```

  (Contributed by Raymond Hettinger.)

- The \texttt{format()} function in the \texttt{locale} module has been modified
and two new functions were added, `format_string()` and `currency()`.

The `format()` function’s `val` parameter could previously be a string as long as no more than one `%` character specifier appeared; now the parameter must be exactly one `%` character specifier with no surrounding text. An optional `monetary` parameter was also added which, if `True`, will use the locale’s rules for formatting currency in placing a separator between groups of three digits.

To format strings with multiple `%` character specifiers, use the new `format_string()` function that works like `format()` but also supports mixing `%` character specifiers with arbitrary text.

A new `currency()` function was also added that formats a number according to the current locale’s settings.

(Contributed by Georg Brandl.)

- The `mailbox` module underwent a massive rewrite to add the capability to modify mailboxes in addition to reading them. A new set of classes that include `mbox`, `MH`, and `Maildir` are used to read mailboxes, and have an `add(message)()` method to add messages, `remove(key)()` to remove messages, and `lock() / unlock()` to lock/unlock the mailbox. The following example converts a maildir-format mailbox into an mbox-format one:

```python
import mailbox

# 'factory=None' uses email.Message.Message as the class representing individual messages.
src = mailbox.Maildir('maildir', factory=None)
dest = mailbox.mbox('/tmp/mbox')

for msg in src:
    dest.add(msg)
```
New module: the `msilib` module allows creating Microsoft Installer `.msi` files and CAB files. Some support for reading the `.msi` database is also included. (Contributed by Martin von Löwis.)

The `nis` module now supports accessing domains other than the system default domain by supplying a `domain` argument to the `nis.match()` and `nis.maps()` functions. (Contributed by Ben Bell.)

The `operator` module's `itemgetter()` and `attrgetter()` functions now support multiple fields. A call such as `operator.attrgetter('a', 'b')` will return a function that retrieves the `a` and `b` attributes. Combining this new feature with the `sort()` method's `key` parameter lets you easily sort lists using multiple fields. (Contributed by Raymond Hettinger.)

The `optparse` module was updated to version 1.5.1 of the Optik library. The `OptionParser` class gained an `epilog` attribute, a string that will be printed after the help message, and a `destroy()` method to break reference cycles created by the object. (Contributed by Greg Ward.)

The `os` module underwent several changes. The `stat_float_times` variable now defaults to true, meaning that `os.stat()` will now return time values as floats. (This doesn't necessarily mean that `os.stat()` will return times that are precise to fractions of a second; not all systems support such
precision.)

Constants named `os.SEEK_SET`, `os.SEEK_CUR`, and `os.SEEK_END` have been added; these are the parameters to the `os.lseek()` function. Two new constants for locking are `os.O_SHLOCK` and `os.O_EXLOCK`.

Two new functions, `wait3()` and `wait4()`, were added. They're similar the `waitpid()` function which waits for a child process to exit and returns a tuple of the process ID and its exit status, but `wait3()` and `wait4()` return additional information. `wait3()` doesn't take a process ID as input, so it waits for any child process to exit and returns a 3-tuple of `process-id`, `exit-status`, `resource-usage` as returned from the `resource.getrusage()` function. `wait4(pid)()` does take a process ID. (Contributed by Chad J. Schroeder.)

On FreeBSD, the `os.stat()` function now returns times with nanosecond resolution, and the returned object now has `st_gen` and `st_birthtime`. The `st_flags` attribute is also available, if the platform supports it. (Contributed by Antti Louko and Diego Pettenò.)

- The Python debugger provided by the `pdb` module can now store lists of commands to execute when a breakpoint is reached and execution stops. Once breakpoint #1 has been created, enter `commands 1` and enter a series of commands to be executed, finishing the list with `end`. The command list can include commands that resume execution, such as `continue` or `next`. (Contributed by Grégoire Dooms.)

- The `pickle` and `cPickle` modules no longer accept a return value of `None` from the `__reduce__()` method; the method must
return a tuple of arguments instead. The ability to return `None` was deprecated in Python 2.4, so this completes the removal of the feature.

- The `pkgutil` module, containing various utility functions for finding packages, was enhanced to support PEP 302’s import hooks and now also works for packages stored in ZIP-format archives. (Contributed by Phillip J. Eby.)

- The `pybench` benchmark suite by Marc-André Lemburg is now included in the `Tools/pybench` directory. The `pybench` suite is an improvement on the commonly used `pystone.py` program because `pybench` provides a more detailed measurement of the interpreter’s speed. It times particular operations such as function calls, tuple slicing, method lookups, and numeric operations, instead of performing many different operations and reducing the result to a single number as `pystone.py` does.

- The `pyexpat` module now uses version 2.0 of the Expat parser. (Contributed by Trent Mick.)

- The `Queue` class provided by the `Queue` module gained two new methods. `join()` blocks until all items in the queue have been retrieved and all processing work on the items have been completed. Worker threads call the other new method, `task_done()`, to signal that processing for an item has been completed. (Contributed by Raymond Hettinger.)

- The old `regex` and `regsub` modules, which have been deprecated ever since Python 2.0, have finally been deleted. Other deleted modules: `statcache`, `tzparse`, `whrandom`.

- Also deleted: the `lib-old` directory, which includes ancient modules such as `dircmp` and `ni`, was removed. `lib-old` wasn’t
on the default `sys.path`, so unless your programs explicitly added the directory to `sys.path`, this removal shouldn’t affect your code.

- The `rlcompleter` module is no longer dependent on importing the `readline` module and therefore now works on non-Unix platforms. (Patch from Robert Kiendl.)

- The `SimpleXMLRPCServer` and `DocXMLRPCServer` classes now have a `rpc_paths` attribute that constrains XML-RPC operations to a limited set of URL paths; the default is to allow only '/\ and '/RPC2'. Setting `rpc_paths` to `None` or an empty tuple disables this path checking.

- The `socket` module now supports `AF_NETLINK` sockets on Linux, thanks to a patch from Philippe Biondi. Netlink sockets are a Linux-specific mechanism for communications between a user-space process and kernel code; an introductory article about them is at https://www.linuxjournal.com/article/7356. In Python code, netlink addresses are represented as a tuple of 2 integers, `(pid, group_mask).

Two new methods on socket objects, `recv_into(buffer)()` and `recvfrom_into(buffer)()`, store the received data in an object that supports the buffer protocol instead of returning the data as a string. This means you can put the data directly into an array or a memory-mapped file.

Socket objects also gained `getfamily()`, `gettype()`, and `getproto()` accessor methods to retrieve the family, type, and protocol values for the socket.

- New module: the `spwd` module provides functions for accessing the shadow password database on systems that support
shadow passwords.

- The `struct` is now faster because it compiles format strings into `struct` objects with `pack()` and `unpack()` methods. This is similar to how the `re` module lets you create compiled regular expression objects. You can still use the module-level `pack()` and `unpack()` functions; they'll create `struct` objects and cache them. Or you can use `struct` instances directly:

```python
s = struct.Struct('ih3s')
data = s.pack(1972, 187, 'abc')
year, number, name = s.unpack(data)
```

You can also pack and unpack data to and from buffer objects directly using the `pack_into(buffer, offset, v1, v2, ...)()` and `unpack_from(buffer, offset)()` methods. This lets you store data directly into an array or a memory-mapped file.

(Struct objects were implemented by Bob Ippolito at the NeedForSpeed sprint. Support for buffer objects was added by Martin Blais, also at the NeedForSpeed sprint.)

- The Python developers switched from CVS to Subversion during the 2.5 development process. Information about the exact build version is available as the `sys.subversion` variable, a 3-tuple of `(interpreter-name, branch-name, revision-range)`. For example, at the time of writing my copy of 2.5 was reporting (`'CPython', 'trunk', '45313:45315').

This information is also available to C extensions via the `Py_GetBuildInfo()` function that returns a string of build information like this: "trunk:45355:45356M, Apr 13 2006, 07:42:19". (Contributed by Barry Warsaw.)
• Another new function, `sys._current_frames()`, returns the current stack frames for all running threads as a dictionary mapping thread identifiers to the topmost stack frame currently active in that thread at the time the function is called. (Contributed by Tim Peters.)

• The `TarFile` class in the `tarfile` module now has an `extractall()` method that extracts all members from the archive into the current working directory. It’s also possible to set a different directory as the extraction target, and to unpack only a subset of the archive’s members.

  The compression used for a tarfile opened in stream mode can now be autodetected using the mode `'r|*'`. (Contributed by Lars Gustäbel.)

• The `threading` module now lets you set the stack size used when new threads are created. The `stack_size([*size*])(` function returns the currently configured stack size, and supplying the optional `size` parameter sets a new value. Not all platforms support changing the stack size, but Windows, POSIX threading, and OS/2 all do. (Contributed by Andrew MacIntyre.)

• The `unicodedata` module has been updated to use version 4.1.0 of the Unicode character database. Version 3.2.0 is required by some specifications, so it’s still available as `unicodedata.ucd_3_2_0`.

• New module: the `uuid` module generates universally unique identifiers (UUIDs) according to RFC 4122. The RFC defines several different UUID versions that are generated from a starting string, from system properties, or purely randomly. This module contains a `UUID` class and functions named `uuid1()`, `uuid3()`, `uuid4()`, and `uuid5()` to generate different versions of
UUID. (Version 2 UUIDs are not specified in RFC 4122 and are not supported by this module.)

```python
>>> import uuid
>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9ae199e5d')
```

(Contributed by Ka-Ping Yee.)

- The **weakref** module's **WeakKeyDictionary** and **WeakValueDictionary** types gained new methods for iterating over the weak references contained in the dictionary. `iterkeyrefs()` and `keyrefs()` methods were added to **WeakKeyDictionary**, and `itervaluerefs()` and `valuerefs()` were added to **WeakValueDictionary**. (Contributed by Fred L. Drake, Jr.)

- The **webbrowser** module received a number of enhancements. It's now usable as a script with `python -m webbrowser`, taking a URL as the argument; there are a number of switches to control the behaviour (`-n` for a new browser window, `-t` for a new tab). New module-level functions, `open_new()` and `open_new_tab()`, were added to support this. The module's `open()` function supports an additional feature, an `autoraise` parameter that
signals whether to raise the open window when possible. A number of additional browsers were added to the supported list such as Firefox, Opera, Konqueror, and elinks. (Contributed by Oleg Broytmann and Georg Brandl.)

- The xmlrpclib module now supports returning datetime objects for the XML-RPC date type. Supply use_datetime=True to the loads() function or the Unmarshaller class to enable this feature. (Contributed by Skip Montanaro.)

- The zipfile module now supports the ZIP64 version of the format, meaning that a .zip archive can now be larger than 4 GiB and can contain individual files larger than 4 GiB. (Contributed by Ronald Oussoren.)

- The zlib module's Compress and Decompress objects now support a copy() method that makes a copy of the object's internal state and returns a new Compress Or Decompress object. (Contributed by Chris AtLee.)

## The ctypes package

The ctypes package, written by Thomas Heller, has been added to the standard library. ctypes lets you call arbitrary functions in shared libraries or DLLs. Long-time users may remember the dl module, which provides functions for loading shared libraries and calling functions in them. The ctypes package is much fancier.

To load a shared library or DLL, you must create an instance of the CDLL class and provide the name or path of the shared library or DLL. Once that’s done, you can call arbitrary functions by accessing them as attributes of the CDLL object.

```
import ctypes
```
Type constructors for the various C types are provided: \texttt{c\_int()}, \texttt{c\_float()}, \texttt{c\_double()}, \texttt{c\_char\_p()} (equivalent to \texttt{char *})

Unlike Python's types, the C versions are all mutable; you can assign to their \texttt{value} attribute to change the wrapped value. Python integers and strings will be automatically converted to the corresponding C types, but for other types you must call the correct type constructor. (And I mean \textit{must}; getting it wrong will often result in the interpreter crashing with a segmentation fault.)

You shouldn't use \texttt{c\_char\_p()} with a Python string when the C function will be modifying the memory area, because Python strings are supposed to be immutable; breaking this rule will cause puzzling bugs. When you need a modifiable memory area, use \texttt{create\_string\_buffer()}:

\begin{verbatim}
s = "this is a string"
buf = ctypes.create_string_buffer(s)
libc.strfry(buf)
\end{verbatim}

C functions are assumed to return integers, but you can set the \texttt{restype} attribute of the function object to change this:

\begin{verbatim}
>>> libc.atof('2.71828')
-1783957616
>>> libc.atof.retype = ctypes.c_double
>>> libc.atof('2.71828')
2.71828
\end{verbatim}

\texttt{ctypes} also provides a wrapper for Python's C API as the \texttt{ctypes.pythonapi} object. This object does \textit{not} release the global interpreter lock before calling a function, because the lock must be held when calling into the interpreter's code. There's a \texttt{py\_object()}
type constructor that will create a **PyObject** * pointer. A simple usage:

```python
import ctypes

d = {}
c ctypes.pythonapi.PyObject_SetItem(ctypes.py_object(d),
    ctypes.py_object("abc"),  ctypes.py_object(1))

# d is now {'abc', 1}.
```

Don't forget to use `py_object()`; if it's omitted you end up with a segmentation fault.

c**types** has been around for a while, but people still write and distribution hand-coded extension modules because you can't rely on **ctype**s being present. Perhaps developers will begin to write Python wrappers atop a library accessed through **ctype**s instead of extension modules, now that **ctype**s is included with core Python.

**See also:**

http://starship.python.net/crew/theller/ctypes/

The ctypes web page, with a tutorial, reference, and FAQ.

The documentation for the **ctype**s module.

**The ElementTree package**

A subset of Fredrik Lundh’s ElementTree library for processing XML has been added to the standard library as `xml.etree`. The available modules are `ElementTree`, `ElementPath`, and `ElementInclude` from ElementTree 1.2.6. The `cElementTree` accelerator module is also included.

The rest of this section will provide a brief overview of using

ElementTree represents an XML document as a tree of element nodes. The text content of the document is stored as the text and tail attributes of (This is one of the major differences between ElementTree and the Document Object Model; in the DOM there are many different types of node, includingTextNode.)

The most commonly used parsing function is parse(), that takes either a string (assumed to contain a filename) or a file-like object and returns an ElementTree instance:

```python
from xml.etree import ElementTree as ET

# Parse an XML file
tree = ET.parse('ex-1.xml')

# Parse an XML feed
feed = urllib.urlopen('http://planet.python.org/rss10.xml')
tree = ET.parse(feed)
```

Once you have an ElementTree instance, you can call its getroot() method to get the root Element node.

There's also an XML() function that takes a string literal and returns an Element node (not an ElementTree). This function provides a tidy way to incorporate XML fragments, approaching the convenience of an XML literal:

```python
text = ET.XML('<!-- <!-- -->')

text.set('height', '320px')
text.append(elem1)
```

Each XML element supports some dictionary-like and some list-like access methods. Dictionary-like operations are used to access attribute values, and list-like operations are used to access child
nodes.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>elem[n]</code></td>
<td>Returns n'th child element.</td>
</tr>
<tr>
<td><code>elem[m:n]</code></td>
<td>Returns list of m'th through n'th child elements.</td>
</tr>
<tr>
<td><code>len(elem)</code></td>
<td>Returns number of child elements.</td>
</tr>
<tr>
<td><code>list(elem)</code></td>
<td>Returns list of child elements.</td>
</tr>
<tr>
<td><code>elem.append(elem2)</code></td>
<td>Adds <code>elem2</code> as a child.</td>
</tr>
<tr>
<td><code>elem.insert(index, elem2)</code></td>
<td>Inserts <code>elem2</code> at the specified location.</td>
</tr>
<tr>
<td><code>del elem[n]</code></td>
<td>Deletes n'th child element.</td>
</tr>
<tr>
<td><code>elem.keys()</code></td>
<td>Returns list of attribute names.</td>
</tr>
<tr>
<td><code>elem.get(name)</code></td>
<td>Returns value of attribute <code>name</code>.</td>
</tr>
<tr>
<td><code>elem.set(name, value)</code></td>
<td>Sets new value for attribute <code>name</code>.</td>
</tr>
<tr>
<td><code>elem.attrib</code></td>
<td>Retrieves the dictionary containing attributes.</td>
</tr>
<tr>
<td><code>del elem.attrib[name]</code></td>
<td>Deletes attribute <code>name</code>.</td>
</tr>
</tbody>
</table>

Comments and processing instructions are also represented as `Element` nodes. To check if a node is a comment or processing instructions:

```python
if elem.tag is ET.Comment:
    ...
elif elem.tag is ET.ProcessingInstruction:
    ...
```

To generate XML output, you should call the `ElementTree.write()` method. Like `parse()`, it can take either a string or a file-like object:

```python
# Encoding is US-ASCII
tree.write('output.xml')

# Encoding is UTF-8
f = open('output.xml', 'w')
tree.write(f, encoding='utf-8')
```
(Caution: the default encoding used for output is ASCII. For general XML work, where an element's name may contain arbitrary Unicode characters, ASCII isn't a very useful encoding because it will raise an exception if an element's name contains any characters with values greater than 127. Therefore, it's best to specify a different encoding such as UTF-8 that can handle any Unicode character.)

This section is only a partial description of the ElementTree interfaces. Please read the package’s official documentation for more details.

See also:

http://effbot.org/zone/element-index.htm

Official documentation for ElementTree.

The hashlib package

A new `hashlib` module, written by Gregory P. Smith, has been added to replace the `md5` and `sha` modules. `hashlib` adds support for additional secure hashes (SHA-224, SHA-256, SHA-384, and SHA-512). When available, the module uses OpenSSL for fast platform optimized implementations of algorithms.

The old `md5` and `sha` modules still exist as wrappers around hashlib to preserve backwards compatibility. The new module's interface is very close to that of the old modules, but not identical. The most significant difference is that the constructor functions for creating new hashing objects are named differently.

```python
# Old versions
h = md5.md5()
h = md5.new()
```
Once a hash object has been created, its methods are the same as before: `update(string)()` hashes the specified string into the current digest state, `digest()` and `hexdigest()` return the digest value as a binary string or a string of hex digits, and `copy()` returns a new hashing object with the same digest state.

**See also:** The documentation for the `hashlib` module.

### The sqlite3 package

The `pysqlite` module (http://www.pysqlite.org), a wrapper for the SQLite embedded database, has been added to the standard library under the package name `sqlite3`.

SQLite is a C library that provides a lightweight disk-based database that doesn’t require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It’s also possible to prototype an application using SQLite
and then port the code to a larger database such as PostgreSQL or Oracle.

pysqlite was written by Gerhard Häring and provides a SQL interface compliant with the DB-API 2.0 specification described by PEP 249.

If you’re compiling the Python source yourself, note that the source tree doesn’t include the SQLite code, only the wrapper module. You'll need to have the SQLite libraries and headers installed before compiling Python, and the build process will compile the module when the necessary headers are available.

To use the module, you must first create a `Connection` object that represents the database. Here the data will be stored in the `/tmp/example` file:

```python
conn = sqlite3.connect('/tmp/example')
```

You can also supply the special name `:memory:` to create a database in RAM.

Once you have a `Connection`, you can create a `Cursor` object and call its `execute()` method to perform SQL commands:

```python
c = conn.cursor()

# Create table
c.execute('''create table stocks
  (date text, trans text, symbol text,
   qty real, price real)''')

# Insert a row of data
insert_into_stocks = ('''insert into stocks
  values ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)'''

c.execute(insert_into_stocks)
```

Usually your SQL operations will need to use values from Python variables. You shouldn’t assemble your query using Python’s string
operations because doing so is insecure; it makes your program vulnerable to an SQL injection attack.

Instead, use the DB-API's parameter substitution. Put ? as a placeholder wherever you want to use a value, and then provide a tuple of values as the second argument to the cursor's execute() method. (Other database modules may use a different placeholder, such as %s or :1.) For example:

```python
# Never do this -- insecure!
symbol = 'IBM'
c.execute("... where symbol = '%s'" % symbol)

# Do this instead
t = (symbol,)
c.execute('select * from stocks where symbol=?', t)

# Larger example
for t in (('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
          ('2006-04-05', 'BUY', 'MSOFT', 1000, 72.00),
          ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
         ):
    c.execute('insert into stocks values (?,?,?,?)', t)
```

To retrieve data after executing a SELECT statement, you can either treat the cursor as an iterator, call the cursor's fetchone() method to retrieve a single matching row, or call fetchall() to get a list of the matching rows.

This example uses the iterator form:

```python
>>> c = conn.cursor()
>>> c.execute('select * from stocks order by price')
>>> for row in c:
...    print row
...
(u'2006-01-05', u'BUY', u'RHAT', 100, 35.140000000000001)
(u'2006-03-28', u'BUY', u'IBM', 1000, 45.0)
(u'2006-04-06', u'SELL', u'IBM', 500, 53.0)
(u'2006-04-05', u'BUY', u'MSOFT', 1000, 72.0)
```
For more information about the SQL dialect supported by SQLite, see https://www.sqlite.org.

See also:

http://www.pysqalite.org
   The pysqlite web page.

https://www.sqlite.org
   The SQLite web page; the documentation describes the syntax and the available data types for the supported SQL dialect.

The documentation for the sqlite3 module.

PEP 249 - Database API Specification 2.0
   PEP written by Marc-André Lemburg.

The wsgiref package

The Web Server Gateway Interface (WSGI) v1.0 defines a standard interface between web servers and Python web applications and is described in PEP 333. The wsgiref package is a reference implementation of the WSGI specification.

The package includes a basic HTTP server that will run a WSGI application; this server is useful for debugging but isn’t intended for production use. Setting up a server takes only a few lines of code:

```python
from wsgiref import simple_server

wsgi_app = ...

host = ''
port = 8000
httpd = simple_server.make_server(host, port, wsgi_app)
```
See also:

http://www.wsgi.org
   A central web site for WSGI-related resources.

PEP 333 - Python Web Server Gateway Interface v1.0
   PEP written by Phillip J. Eby.
Build and C API Changes

Changes to Python’s build process and to the C API include:

- The Python source tree was converted from CVS to Subversion, in a complex migration procedure that was supervised and flawlessly carried out by Martin von Löwis. The procedure was developed as PEP 347.

- Coverity, a company that markets a source code analysis tool called Prevent, provided the results of their examination of the Python source code. The analysis found about 60 bugs that were quickly fixed. Many of the bugs were refcounting problems, often occurring in error-handling code. See https://scan.coverity.com for the statistics.

- The largest change to the C API came from PEP 353, which modifies the interpreter to use a `Py_ssize_t` type definition instead of `int`. See the earlier section PEP 353: Using ssize_t as the index type for a discussion of this change.

- The design of the bytecode compiler has changed a great deal, no longer generating bytecode by traversing the parse tree. Instead the parse tree is converted to an abstract syntax tree (or AST), and it is the abstract syntax tree that’s traversed to produce the bytecode.

It’s possible for Python code to obtain AST objects by using the `compile()` built-in and specifying `_ast.PyCF_ONLY_AST` as the value of the `flags` parameter:

```python
from _ast import PyCF_ONLY_AST
ast = compile("""a=0
for i in range(10):
    a += i
"""), PyCF_ONLY_AST)
```
No official documentation has been written for the AST code yet, but PEP 339 discusses the design. To start learning about the code, read the definition of the various AST nodes in Parser/Python.asdl. A Python script reads this file and generates a set of C structure definitions in Include/Python-ast.h. The PyParser_ASTFromString() and PyParser_ASTFromFile(), defined in Include/pythonrun.h, take Python source as input and return the root of an AST representing the contents. This AST can then be turned into a code object by PyAST_Compile(). For more information, read the source code, and then ask questions on python-dev.

The AST code was developed under Jeremy Hylton's management, and implemented by (in alphabetical order) Brett Cannon, Nick Coghlan, Grant Edwards, John Ehresman, Kurt Kaiser, Neal Norwitz, Tim Peters, Armin Rigo, and Neil Schemenauer, plus the participants in a number of AST sprints at conferences such as PyCon.

- Evan Jones's patch to obmalloc, first described in a talk at PyCon DC 2005, was applied. Python 2.4 allocated small objects in 256K-sized arenas, but never freed arenas. With this patch, Python will free arenas when they're empty. The net effect is that on some platforms, when you allocate many objects, Python's memory usage may actually drop when you delete them and the memory may be returned to the operating system. (Implemented by Evan Jones, and reworked by Tim Peters.)

Note that this change means extension modules must be more
careful when allocating memory. Python’s API has many different functions for allocating memory that are grouped into families. For example, \texttt{PyMem\_Malloc()}, \texttt{PyMem\_Realloc()}, and \texttt{PyMem\_Free()} are one family that allocates raw memory, while \texttt{PyObject\_Malloc()}, \texttt{PyObject\_Realloc()}, and \texttt{PyObject\_Free()} are another family that’s supposed to be used for creating Python objects.

Previously these different families all reduced to the platform’s malloc() and free() functions. This meant it didn’t matter if you got things wrong and allocated memory with the \texttt{PyMem()} function but freed it with the \texttt{PyObject()} function. With 2.5’s changes to obmalloc, these families now do different things and mismatches will probably result in a segfault. You should carefully test your C extension modules with Python 2.5.

- The built-in set types now have an official C API. Call \texttt{PySet\_New()} and \texttt{PyFrozenSet\_New()} to create a new set, \texttt{PySet\_Add()} and \texttt{PySet\_Discard()} to add and remove elements, and \texttt{PySet\_Contains()} and \texttt{PySet\_Size()} to examine the set’s state. (Contributed by Raymond Hettinger.)

- C code can now obtain information about the exact revision of the Python interpreter by calling the \texttt{Py\_GetBuildInfo()} function that returns a string of build information like this: 

- Two new macros can be used to indicate C functions that are local to the current file so that a faster calling convention can be used. \texttt{Py\_LOCAL(type)()} declares the function as returning a value of the specified type and uses a fast-calling qualifier. \texttt{Py\_LOCAL\_INLINE(type)()} does the same thing and also
requests the function be inlined. If \texttt{PY_LOCAL_AGGRESSIVE()} is defined before \texttt{python.h} is included, a set of more aggressive optimizations are enabled for the module; you should benchmark the results to find out if these optimizations actually make the code faster. (Contributed by Fredrik Lundh at the NeedForSpeed sprint.)

- \texttt{PyErr\_NewException(name, base, dict)()} can now accept a tuple of base classes as its \textit{base} argument. (Contributed by Georg Brandl.)

- The \texttt{PyErr\_Warn()} function for issuing warnings is now deprecated in favour of \texttt{PyErr\_WarnEx(category, message, stacklevel)()} which lets you specify the number of stack frames separating this function and the caller. A \textit{stacklevel} of 1 is the function calling \texttt{PyErr\_WarnEx()}, 2 is the function above that, and so forth. (Added by Neal Norwitz.)

- The CPython interpreter is still written in C, but the code can now be compiled with a C++ compiler without errors. (Implemented by Anthony Baxter, Martin von Löwis, Skip Montanaro.)

- The \texttt{PyRange\_New()} function was removed. It was never documented, never used in the core code, and had dangerously lax error checking. In the unlikely case that your extensions were using it, you can replace it by something like the following:

```python
range = PyObject\_CallFunction((PyObject*) &PyRange\_Type, "lll", start, stop, step);
```

### Port-Specific Changes

- MacOS X (10.3 and higher): dynamic loading of modules now
uses the `dlopen()` function instead of MacOS-specific functions.

- MacOS X: an `--enable-universal-sdk` switch was added to the `configure` script that compiles the interpreter as a universal binary able to run on both PowerPC and Intel processors. (Contributed by Ronald Oussoren; issue 2573.)
- Windows: `.dll` is no longer supported as a filename extension for extension modules. `.pyd` is now the only filename extension that will be searched for.
This section lists previously described changes that may require changes to your code:

- ASCII is now the default encoding for modules. It’s now a syntax error if a module contains string literals with 8-bit characters but doesn’t have an encoding declaration. In Python 2.4 this triggered a warning, not a syntax error.
- Previously, the `gi_frame` attribute of a generator was always a frame object. Because of the PEP 342 changes described in section PEP 342: New Generator Features, it’s now possible for `gi_frame` to be `None`.
- A new warning, `UnicodeWarning`, is triggered when you attempt to compare a Unicode string and an 8-bit string that can’t be converted to Unicode using the default ASCII encoding. Previously such comparisons would raise a `UnicodeDecodeError` exception.
- Library: the `csv` module is now stricter about multi-line quoted fields. If your files contain newlines embedded within fields, the input should be split into lines in a manner which preserves the newline characters.
- Library: the `locale` module’s `format()` function’s would previously accept any string as long as no more than one `%char` specifier appeared. In Python 2.5, the argument must be exactly one `%char` specifier with no surrounding text.
- Library: The `pickle` and `cPickle` modules no longer accept a return value of `None` from the `__reduce__()` method; the method must return a tuple of arguments instead. The modules also no longer accept the deprecated `bin` keyword parameter.
- Library: The `SimpleXMLRPCServer` and `DocXMLRPCServer` classes now have a `rpc_paths` attribute that constrains XML-RPC
operations to a limited set of URL paths; the default is to allow only '//' and '/RPC2'. Setting `rpc_paths` to `None` or an empty tuple disables this path checking.

- **C API:** Many functions now use `Py_ssize_t` instead of `int` to allow processing more data on 64-bit machines. Extension code may need to make the same change to avoid warnings and to support 64-bit machines. See the earlier section [PEP 353: Using ssize_t as the index type](https://www.python.org/dev/peps/pep-0353/) for a discussion of this change.

- **C API:** The obmalloc changes mean that you must be careful to not mix usage of the `PyMem_*()` and `PyObject_*()` families of functions. Memory allocated with one family's `*_Malloc()` must be freed with the corresponding family's `*_Free()` function.
Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Georg Brandl, Nick Coghlan, Phillip J. Eby, Lars Gustäbel, Raymond Hettinger, Ralf W. Grosse-Kunstleve, Kent Johnson, Iain Lowe, Martin von Löwis, Fredrik Lundh, Andrew McNamara, Skip Montanaro, Gustavo Niemeyer, Paul Prescod, James Pryor, Mike Rovner, Scott Weikart, Barry Warsaw, Thomas Wouters.
This article explains the new features in Python 2.4.1, released on March 30, 2005.

Python 2.4 is a medium-sized release. It doesn’t introduce as many changes as the radical Python 2.2, but introduces more features than the conservative 2.3 release. The most significant new language features are function decorators and generator expressions; most other changes are to the standard library.

According to the CVS change logs, there were 481 patches applied and 502 bugs fixed between Python 2.3 and 2.4. Both figures are likely to be underestimates.

This article doesn’t attempt to provide a complete specification of every single new feature, but instead provides a brief introduction to each feature. For full details, you should refer to the documentation for Python 2.4, such as the Python Library Reference and the Python Reference Manual. Often you will be referred to the PEP for a particular new feature for explanations of the implementation and design rationale.
PEP 218: Built-In Set Objects

Python 2.3 introduced the \texttt{sets} module. C implementations of set data types have now been added to the Python core as two new built-in types, \texttt{set(iterable)()} and \texttt{frozenset(iterable)()}. They provide high speed operations for membership testing, for eliminating duplicates from sequences, and for mathematical operations like unions, intersections, differences, and symmetric differences.

\begin{verbatim}
>>> a = set('abracadabra')     # form a set from a string
>>> 'z' in a                  # fast membership testing
False
>>> a                         # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> ''.join(a)                # convert back into a string
'arbcd'

>>> b = set('alacazam')       # form a second set
>>> a - b                     # letters in a but not b
set(['r', 'd', 'b'])
>>> a | b                     # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                     # letters in both a and b
set(['a', 'c'])
>>> a ^ b                     # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])

>>> a.add('z')                # add a new element
>>> a.update('wxy')           # add multiple new elements
>>> a                         # result
set(['a', 'c', 'b', 'd', 'r', 'w', 'y', 'x', 'z'])
>>> a.remove('x')             # take one element out
>>> a                         # result
set(['a', 'c', 'b', 'd', 'r', 'w', 'y', 'z'])
\end{verbatim}

The \texttt{frozenset()} type is an immutable version of \texttt{set()}. Since it is immutable and hashable, it may be used as a dictionary key or as a member of another set.
The `sets` module remains in the standard library, and may be useful if you wish to subclass the `Set` or `ImmutableSet` classes. There are currently no plans to deprecate the module.

**See also:**

**PEP 218 - Adding a Built-In Set Object Type**

Originally proposed by Greg Wilson and ultimately implemented by Raymond Hettinger.
PEP 237: Unifying Long Integers and Integers

The lengthy transition process for this PEP, begun in Python 2.2, takes another step forward in Python 2.4. In 2.3, certain integer operations that would behave differently after int/long unification triggered `FutureWarning` warnings and returned values limited to 32 or 64 bits (depending on your platform). In 2.4, these expressions no longer produce a warning and instead produce a different result that's usually a long integer.

The problematic expressions are primarily left shifts and lengthy hexadecimal and octal constants. For example, `2 << 32` results in a warning in 2.3, evaluating to 0 on 32-bit platforms. In Python 2.4, this expression now returns the correct answer, 8589934592.

See also:

PEP 237 - Unifying Long Integers and Integers

Original PEP written by Moshe Zadka and GvR. The changes for 2.4 were implemented by Kalle Svensson.
PEP 289: Generator Expressions

The iterator feature introduced in Python 2.2 and the `itertools` module make it easier to write programs that loop through large data sets without having the entire data set in memory at one time. List comprehensions don’t fit into this picture very well because they produce a Python list object containing all of the items. This unavoidably pulls all of the objects into memory, which can be a problem if your data set is very large. When trying to write a functionally-styled program, it would be natural to write something like:

```python
links = [link for link in get_all_links() if not link.followed]
for link in links:
    ...
```

instead of

```python
for link in get_all_links():
    if link.followed:
        continue
    ...
```

The first form is more concise and perhaps more readable, but if you're dealing with a large number of link objects you’d have to write the second form to avoid having all link objects in memory at the same time.

Generator expressions work similarly to list comprehensions but don’t materialize the entire list; instead they create a generator that will return elements one by one. The above example could be written as:

```python
links = (link for link in get_all_links() if not link.followed)
for link in links:
```
Generator expressions always have to be written inside parentheses, as in the above example. The parentheses signalling a function call also count, so if you want to create an iterator that will be immediately passed to a function you could write:

```
print sum(obj.count for obj in list_all_objects())
```

Generator expressions differ from list comprehensions in various small ways. Most notably, the loop variable (`obj` in the above example) is not accessible outside of the generator expression. List comprehensions leave the variable assigned to its last value; future versions of Python will change this, making list comprehensions match generator expressions in this respect.

**See also:**

**PEP 289 - Generator Expressions**

Proposed by Raymond Hettinger and implemented by Jiwon Seo with early efforts steered by Hye-Shik Chang.
PEP 292: Simpler String Substitutions

Some new classes in the standard library provide an alternative mechanism for substituting variables into strings; this style of substitution may be better for applications where untrained users need to edit templates.

The usual way of substituting variables by name is the `%` operator:

```python
>>> '%(page)i: %(title)s' % {'page':2, 'title': 'The Best of Times'}
'2: The Best of Times'
```

When writing the template string, it can be easy to forget the `i` or `s` after the closing parenthesis. This isn’t a big problem if the template is in a Python module, because you run the code, get an “Unsupported format character” `ValueError`, and fix the problem. However, consider an application such as Mailman where template strings or translations are being edited by users who aren’t aware of the Python language. The format string’s syntax is complicated to explain to such users, and if they make a mistake, it’s difficult to provide helpful feedback to them.

PEP 292 adds a `Template` class to the `string` module that uses `$` to indicate a substitution:

```python
>>> import string
>>> t = string.Template('$page: $title')
>>> t.substitute({'page':2, 'title': 'The Best of Times'})
'2: The Best of Times'
```

If a key is missing from the dictionary, the `substitute()` method will raise a `KeyError`. There’s also a `safe_substitute()` method that ignores missing keys:
>>> t = string.Template('page: $title')
>>> t.safe_substitute({'page': 3})
'3: $title'

See also:

**PEP 292 - Simpler String Substitutions**
Written and implemented by Barry Warsaw.
PEP 318: Decorators for Functions and Methods

Python 2.2 extended Python’s object model by adding static methods and class methods, but it didn’t extend Python’s syntax to provide any new way of defining static or class methods. Instead, you had to write a `def` statement in the usual way, and pass the resulting method to a `staticmethod()` or `classmethod()` function that would wrap up the function as a method of the new type. Your code would look like this:

```python
class C:
    def meth (cls):
        ...

    meth = classmethod(meth)  # Rebind name to wrapped-up class
```

If the method was very long, it would be easy to miss or forget the `classmethod()` invocation after the function body.

The intention was always to add some syntax to make such definitions more readable, but at the time of 2.2’s release a good syntax was not obvious. Today a good syntax still isn’t obvious but users are asking for easier access to the feature; a new syntactic feature has been added to meet this need.

The new feature is called “function decorators”. The name comes from the idea that `classmethod()`, `staticmethod()`, and friends are storing additional information on a function object; they’re decorating functions with more details.

The notation borrows from Java and uses the `@` character as an indicator. Using the new syntax, the example above would be
written:

```python
class C:
    @classmethod
def meth(cls):
    ...
```

The `@classmethod` is shorthand for the `meth=classmethod(meth)` assignment. More generally, if you have the following:

```python
@A
@B
@C
def f():
    ...
```

It’s equivalent to the following pre-decorator code:

```python
def f(): ...
f = A(B(C(f)))
```

Decorators must come on the line before a function definition, one decorator per line, and can’t be on the same line as the `def` statement, meaning that `@A def f(): ...` is illegal. You can only decorate function definitions, either at the module level or inside a class; you can’t decorate class definitions.

A decorator is just a function that takes the function to be decorated as an argument and returns either the same function or some new object. The return value of the decorator need not be callable (though it typically is), unless further decorators will be applied to the result. It’s easy to write your own decorators. The following simple example just sets an attribute on the function object:

```python
>>> def deco(func):
    ...
    func.attr = 'decorated'
    ...
    return func
```
As a slightly more realistic example, the following decorator checks that the supplied argument is an integer:

```python
def require_int (func):
    def wrapper (arg):
        assert isinstance(arg, int)
        return func(arg)
    return wrapper

@require_int
def p1 (arg):
    print arg

@require_int
def p2 (arg):
    print arg*2
```

An example in **PEP 318** contains a fancier version of this idea that lets you both specify the required type and check the returned type.

Decorator functions can take arguments. If arguments are supplied, your decorator function is called with only those arguments and must return a new decorator function; this function must take a single function and return a function, as previously described. In other words, `@A @B @C(args)` becomes:

```python
def f(): ...
    _deco = C(args)
    f = A(B(_deco(f)))
```
Getting this right can be slightly brain-bending, but it's not too difficult.

A small related change makes the `func_name` attribute of functions writable. This attribute is used to display function names in tracebacks, so decorators should change the name of any new function that's constructed and returned.

See also:

**PEP 318 - Decorators for Functions, Methods and Classes**

Written by Kevin D. Smith, Jim Jewett, and Skip Montanaro. Several people wrote patches implementing function decorators, but the one that was actually checked in was patch #979728, written by Mark Russell.

[https://wiki.python.org/moin/PythonDecoratorLibrary](https://wiki.python.org/moin/PythonDecoratorLibrary)

This Wiki page contains several examples of decorators.
A new built-in function, `reversed(seq)()`, takes a sequence and returns an iterator that loops over the elements of the sequence in reverse order.

```python
>>> for i in reversed(xrange(1,4)):
...     print i
... 3 2 1
```

Compared to extended slicing, such as `range(1,4)[::-1]`, `reversed()` is easier to read, runs faster, and uses substantially less memory.

Note that `reversed()` only accepts sequences, not arbitrary iterators. If you want to reverse an iterator, first convert it to a list with `list()`.

```python
>>> input = open('/etc/passwd', 'r')
>>> for line in reversed(list(input)):
...     print line
... root:*:0:0:System Administrator:/var/root:/bin/tcsh
...```

See also:

**PEP 322 - Reverse Iteration**

Written and implemented by Raymond Hettinger.
PEP 324: New subprocess Module

The standard library provides a number of ways to execute a subprocess, offering different features and different levels of complexity. `os.system(command)()` is easy to use, but slow (it runs a shell process which executes the command) and dangerous (you have to be careful about escaping the shell’s metacharacters). The `popen2` module offers classes that can capture standard output and standard error from the subprocess, but the naming is confusing. The `subprocess` module cleans this up, providing a unified interface that offers all the features you might need.

Instead of `popen2`’s collection of classes, `subprocess` contains a single class called `Popen` whose constructor supports a number of different keyword arguments.

```python
class Popen(args, bufsize=0, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=False, shell=False, cwd=None, env=None, universal_newlines=False, startupinfo=None, creationflags=0):
```

`args` is commonly a sequence of strings that will be the arguments to the program executed as the subprocess. (If the `shell` argument is true, `args` can be a string which will then be passed on to the shell for interpretation, just as `os.system()` does.)

`stdin`, `stdout`, and `stderr` specify what the subprocess's input, output, and error streams will be. You can provide a file object or a file descriptor, or you can use the constant `subprocess.PIPE` to create a pipe between the subprocess and the parent.

The constructor has a number of handy options:
- `close_fds` requests that all file descriptors be closed before running the subprocess.
- `cwd` specifies the working directory in which the subprocess will be executed (defaulting to whatever the parent’s working directory is).
- `env` is a dictionary specifying environment variables.
- `preexec_fn` is a function that gets called before the child is started.
- `universal_newlines` opens the child’s input and output using Python’s `universal newlines` feature.

Once you’ve created the `Popen` instance, you can call its `wait()` method to pause until the subprocess has exited, `poll()` to check if it’s exited without pausing, or `communicate(data)()` to send the string `data` to the subprocess’s standard input. `communicate(data)()` then reads any data that the subprocess has sent to its standard output or standard error, returning a tuple `(stdout_data, stderr_data)`.

`call()` is a shortcut that passes its arguments along to the `Popen` constructor, waits for the command to complete, and returns the status code of the subprocess. It can serve as a safer analog to `os.system()`:

```python
sts = subprocess.call(['dpkg', '-i', '/tmp/new-package.deb'])
if sts == 0:
    # Success
    ...
else:
    # dpkg returned an error
    ...
```

The command is invoked without use of the shell. If you really do want to use the shell, you can add `shell=True` as a keyword argument and provide a string instead of a sequence:
The PEP takes various examples of shell and Python code and shows how they’d be translated into Python code that uses `subprocess`. Reading this section of the PEP is highly recommended.

See also:

**PEP 324 - subprocess - New process module**

Written and implemented by Peter Åstrand, with assistance from Fredrik Lundh and others.
PEP 327: Decimal Data Type

Python has always supported floating-point (FP) numbers, based on the underlying C `double` type, as a data type. However, while most programming languages provide a floating-point type, many people (even programmers) are unaware that floating-point numbers don’t represent certain decimal fractions accurately. The new `Decimal` type can represent these fractions accurately, up to a user-specified precision limit.

Why is Decimal needed?

The limitations arise from the representation used for floating-point numbers. FP numbers are made up of three components:

- The sign, which is positive or negative.
- The mantissa, which is a single-digit binary number followed by a fractional part. For example, `1.01` in base-2 notation is `1 + 0/2 + 1/4`, or 1.25 in decimal notation.
- The exponent, which tells where the decimal point is located in the number represented.

For example, the number 1.25 has positive sign, a mantissa value of 1.01 (in binary), and an exponent of 0 (the decimal point doesn’t need to be shifted). The number 5 has the same sign and mantissa, but the exponent is 2 because the mantissa is multiplied by 4 (2 to the power of the exponent 2); 1.25 * 4 equals 5.

Modern systems usually provide floating-point support that conforms to a standard called IEEE 754. C’s `double` type is usually implemented as a 64-bit IEEE 754 number, which uses 52 bits of space for the mantissa. This means that numbers can only be specified to 52 bits of precision. If you’re trying to represent numbers
whose expansion repeats endlessly, the expansion is cut off after 52 bits. Unfortunately, most software needs to produce output in base 10, and common fractions in base 10 are often repeating decimals in binary. For example, 1.1 decimal is binary \( 1.0001100110011 \ldots \); \( .1 = 1/16 + 1/32 + 1/256 \) plus an infinite number of additional terms. IEEE 754 has to chop off that infinitely repeated decimal after 52 digits, so the representation is slightly inaccurate.

Sometimes you can see this inaccuracy when the number is printed:

```
>>> 1.1
1.1000000000000001
```

The inaccuracy isn’t always visible when you print the number because the FP-to-decimal-string conversion is provided by the C library, and most C libraries try to produce sensible output. Even if it’s not displayed, however, the inaccuracy is still there and subsequent operations can magnify the error.

For many applications this doesn’t matter. If I’m plotting points and displaying them on my monitor, the difference between 1.1 and 1.1000000000000001 is too small to be visible. Reports often limit output to a certain number of decimal places, and if you round the number to two or three or even eight decimal places, the error is never apparent. However, for applications where it does matter, it’s a lot of work to implement your own custom arithmetic routines.

Hence, the `Decimal` type was created.

### The `Decimal` type

A new module, `decimal`, was added to Python’s standard library. It contains two classes, `Decimal` and `Context`. `Decimal` instances represent numbers, and `Context` instances are used to wrap up various settings such as the precision and default rounding mode.
Decimal instances are immutable, like regular Python integers and FP numbers; once it’s been created, you can’t change the value an instance represents. Decimal instances can be created from integers or strings:

```python
>>> import decimal
>>> decimal.Decimal(1972)
Decimal("1972")
>>> decimal.Decimal("1.1")
Decimal("1.1")
```

You can also provide tuples containing the sign, the mantissa represented as a tuple of decimal digits, and the exponent:

```python
>>> decimal.Decimal((1, (1, 4, 7, 5), -2))
Decimal("-14.75")
```

Cautionary note: the sign bit is a Boolean value, so 0 is positive and 1 is negative.

Converting from floating-point numbers poses a bit of a problem: should the FP number representing 1.1 turn into the decimal number for exactly 1.1, or for 1.1 plus whatever inaccuracies are introduced? The decision was to dodge the issue and leave such a conversion out of the API. Instead, you should convert the floating-point number into a string using the desired precision and pass the string to the Decimal constructor:

```python
>>> f = 1.1
>>> decimal.Decimal(str(f))
Decimal("1.1")
>>> decimal.Decimal('%1.12f' % f)
Decimal("1.100000000000")
```

Once you have Decimal instances, you can perform the usual mathematical operations on them. One limitation: exponentiation requires an integer exponent:
>>> a = decimal.Decimal('35.72')
>>> b = decimal.Decimal('1.73')
>>> a+b
Decimal("37.45")
>>> a-b
Decimal("33.99")
>>> a*b
Decimal("61.7956")
>>> a/b
Decimal("20.6479884393063583815028902")
>>> a**2
Decimal("1275.9184")
>>> a**b
Traceback (most recent call last):
  ...  
decimal.InvalidOperation: x ** (non-integer)

You can combine Decimal instances with integers, but not with floating-point numbers:

>>> a + 4
Decimal("39.72")
>>> a + 4.5
Traceback (most recent call last):
  ...  
TypeError: You can interact Decimal only with int, long or Decimal data types.

Decimal numbers can be used with the math and cmath modules, but note that they'll be immediately converted to floating-point numbers before the operation is performed, resulting in a possible loss of precision and accuracy. You'll also get back a regular floating-point number and not a Decimal.

>>> import math, cmath
>>> d = decimal.Decimal('123456789012.345')
>>> math.sqrt(d)
351364.1828820134
>>> cmath.sqrt(-d)
351364.18288201344j
Decimal instances have a sqrt() method that returns a Decimal, but if you need other things such as trigonometric functions you’ll have to implement them.

```python
>>> d.sqrt()
Decimal("351364.1828820134592177245001")
```

The context type

Instances of the Context class encapsulate several settings for decimal operations:

- **prec** is the precision, the number of decimal places.
- **rounding** specifies the rounding mode. The **decimal** module has constants for the various possibilities: **ROUND_DOWN**, **ROUND_CEILING**, **ROUND_HALF_EVEN**, and various others.
- **traps** is a dictionary specifying what happens on encountering certain error conditions: either an exception is raised or a value is returned. Some examples of error conditions are division by zero, loss of precision, and overflow.

There’s a thread-local default context available by calling `getcontext()`; you can change the properties of this context to alter the default precision, rounding, or trap handling. The following example shows the effect of changing the precision of the default context:

```python
>>> decimal.getcontext().prec
28
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.1428571428571428571428571429")
>>> decimal.getcontext().prec = 9
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.142857143")
```

The default action for error conditions is selectable; the module can
either return a special value such as infinity or not-a-number, or
extceptions can be raised:

```python
>>> decimal.Decimal(1) / decimal.Decimal(0)
Traceback (most recent call last):
  ...  
decimal.DivisionByZero: x / 0
>>> decimal.getcontext().traps[decimal.DivisionByZero] = False
>>> decimal.Decimal(1) / decimal.Decimal(0)
Decimal("Infinity")
```

The `Context` instance also has various methods for formatting
numbers such as `to_eng_string()` and `to_sci_string()`.

For more information, see the documentation for the `decimal`
module, which includes a quick-start tutorial and a reference.

See also:

**PEP 327 - Decimal Data Type**

Written by Facundo Batista and implemented by Facundo

[http://www.lahey.com/float.htm](http://www.lahey.com/float.htm)

The article uses Fortran code to illustrate many of the problems
that floating- point inaccuracy can cause.


A description of a decimal-based representation. This
representation is being proposed as a standard, and underlies
the new Python decimal type. Much of this material was written
by Mike Cowlishaw, designer of the Rexx language.
PEP 328: Multi-line Imports

One language change is a small syntactic tweak aimed at making it easier to import many names from a module. In a `from module import names` statement, `names` is a sequence of names separated by commas. If the sequence is very long, you can either write multiple imports from the same module, or you can use backslashes to escape the line endings like this:

```python
from SimpleXMLRPCServer import SimpleXMLRPCServer,
    SimpleXMLRPCRequestHandler,
    CGIXMLRPCRequestHandler,
    resolve_dotted_attribute
```

The syntactic change in Python 2.4 simply allows putting the names within parentheses. Python ignores newlines within a parenthesized expression, so the backslashes are no longer needed:

```python
from SimpleXMLRPCServer import (SimpleXMLRPCServer,
    SimpleXMLRPCRequestHandler,
    CGIXMLRPCRequestHandler,
    resolve_dotted_attribute)
```

The PEP also proposes that all `import` statements be absolute imports, with a leading `.` character to indicate a relative import. This part of the PEP was not implemented for Python 2.4, but was completed for Python 2.5.

See also:

PEP 328 - Imports: Multi-Line and Absolute/Relative

Written by Aahz. Multi-line imports were implemented by Dima Dorfman.
PEP 331: Locale-Independent Float/String Conversions

The `locale` modules lets Python software select various conversions and display conventions that are localized to a particular country or language. However, the module was careful to not change the numeric locale because various functions in Python’s implementation required that the numeric locale remain set to the 'c' locale. Often this was because the code was using the C library's `atof()` function.

Not setting the numeric locale caused trouble for extensions that used third-party C libraries, however, because they wouldn’t have the correct locale set. The motivating example was GTK+, whose user interface widgets weren’t displaying numbers in the current locale.

The solution described in the PEP is to add three new functions to the Python API that perform ASCII-only conversions, ignoring the locale setting:

- `PyOS_ascii_strtod(str, ptr)()` and `PyOS_ascii_atof(str, ptr)()` both convert a string to a C `double`.
- `PyOS_ascii_formatd(buffer, buf_len, format, d)()` converts a `double` to an ASCII string.

The code for these functions came from the GLib library (https://developer.gnome.org/glib/stable/), whose developers kindly relicensed the relevant functions and donated them to the Python Software Foundation. The `locale` module can now change the numeric locale, letting extensions such as GTK+ produce the correct results.
See also:

**PEP 331** - Locale-Independent Float/String Conversions

Written by Christian R. Reis, and implemented by Gustavo Carneiro.
Other Language Changes

Here are all of the changes that Python 2.4 makes to the core Python language.

- Decorators for functions and methods were added (PEP 318).

- Built-in `set()` and `frozenset()` types were added (PEP 218). Other new built-ins include the `reversed(seq)()` function (PEP 322).

- Generator expressions were added (PEP 289).

- Certain numeric expressions no longer return values restricted to 32 or 64 bits (PEP 237).

- You can now put parentheses around the list of names in a `from module import names` statement (PEP 328).

- The `dict.update()` method now accepts the same argument forms as the `dict` constructor. This includes any mapping, any iterable of key/value pairs, and keyword arguments. (Contributed by Raymond Hettinger.)

- The string methods `ljust()`, `rjust()`, and `center()` now take an optional argument for specifying a fill character other than a space. (Contributed by Raymond Hettinger.)

- Strings also gained an `rsplit()` method that works like the `split()` method but splits from the end of the string. (Contributed by Sean Reifschneider.)

```python
>>> 'www.python.org'.split('.', 1)
['www', 'python.org']
```
Three keyword parameters, `cmp`, `key`, and `reverse`, were added to the `sort()` method of lists. These parameters make some common usages of `sort()` simpler. All of these parameters are optional.

For the `cmp` parameter, the value should be a comparison function that takes two parameters and returns -1, 0, or +1 depending on how the parameters compare. This function will then be used to sort the list. Previously this was the only parameter that could be provided to `sort()`.

`key` should be a single-parameter function that takes a list element and returns a comparison key for the element. The list is then sorted using the comparison keys. The following example sorts a list case-insensitively:

```python
>>> L = ['A', 'b', 'c', 'D']
>>> L.sort()  # Case-sensitive sort
>>> L
['A', 'D', 'b', 'c']
>>> # Using 'key' parameter to sort list
>>> L.sort(key=lambda x: x.lower())
>>> L
['A', 'b', 'c', 'D']
>>> # Old-fashioned way
>>> L.sort(cmp=lambda x, y: cmp(x.lower(), y.lower()))
>>> L
['A', 'b', 'c', 'D']
```

The last example, which uses the `cmp` parameter, is the old way to perform a case-insensitive sort. It works but is slower than using a `key` parameter. Using `key` calls `lower()` method once for each element in the list while using `cmp` will call it twice for each comparison, so using `key` saves on invocations of the `lower()` method.
For simple key functions and comparison functions, it is often possible to avoid a `lambda` expression by using an unbound method instead. For example, the above case-insensitive sort is best written as:

```python
>>> L.sort(key=str.lower)
>>> L
['A', 'b', 'c', 'D']
```

Finally, the `reverse` parameter takes a Boolean value. If the value is true, the list will be sorted into reverse order. Instead of `L.sort(); L.reverse()`, you can now write `L.sort(reverse=True)`.

The results of sorting are now guaranteed to be stable. This means that two entries with equal keys will be returned in the same order as they were input. For example, you can sort a list of people by name, and then sort the list by age, resulting in a list sorted by age where people with the same age are in namesorted order.

(All changes to `sort()` contributed by Raymond Hettinger.)

- There is a new built-in function `sorted(iterable)()` that works like the in-place `list.sort()` method but can be used in expressions. The differences are:
  - the input may be any iterable;
  - a newly formed copy is sorted, leaving the original intact; and
  - the expression returns the new sorted copy

```python
>>> L = [9, 7, 8, 3, 2, 4, 1, 6, 5]
>>> [10+i for i in sorted(L)]  # usable in a list comp
[11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> L  # original is left unch
[9, 7, 8, 3, 2, 4, 1, 6, 5]
```
[9, 7, 8, 3, 2, 4, 1, 6, 5]

```python
>>> sorted('Monty Python') # any iterable may be a
[' ', 'M', 'P', 'h', 'n', 'o', 'o', 't', 't', 'y', 'y'

>>> # List the contents of a dict sorted by key values
>>> colormap = dict(red=1, blue=2, green=3, black=4, yellow=5)
>>> for k, v in sorted(colormap.items()):
...     print k, v
...
black 4
blue 2
green 3
red 1
yellow 5
```

(Contributed by Raymond Hettinger.)

- Integer operations will no longer trigger an `OverflowWarning`. The `OverflowWarning` warning will disappear in Python 2.5.

- The interpreter gained a new switch, `-m`, that takes a name, searches for the corresponding module on `sys.path`, and runs the module as a script. For example, you can now run the Python profiler with `python -m profile`. (Contributed by Nick Coghlan.)

- The `eval(expr, globals, locals)()` and `execfile(filename, globals, locals)()` functions and the `exec` statement now accept any mapping type for the `locals` parameter. Previously this had to be a regular Python dictionary. (Contributed by Raymond Hettinger.)

- The `zip()` built-in function and `itertools.izip()` now return an empty list if called with no arguments. Previously they raised a `TypeError` exception. This makes them more suitable for use with variable length argument lists:
>>> def transpose(array):
...     return zip(*array)
...
>>> transpose([(1, 2, 3), (4, 5, 6)])
[(1, 4), (2, 5), (3, 6)]
>>> transpose([])
[]

(Contributed by Raymond Hettinger.)

- Encountering a failure while importing a module no longer leaves a partially-initialized module object in `sys.modules`. The incomplete module object left behind would fool further imports of the same module into succeeding, leading to confusing errors. (Fixed by Tim Peters.)

- `None` is now a constant; code that binds a new value to the name `None` is now a syntax error. (Contributed by Raymond Hettinger.)

### Optimizations

- The inner loops for list and tuple slicing were optimized and now run about one-third faster. The inner loops for dictionaries were also optimized, resulting in performance boosts for `keys()`, `values()`, `items()`, `iterkeys()`, `itervalues()`, and `iteritems()`. (Contributed by Raymond Hettinger.)

- The machinery for growing and shrinking lists was optimized for speed and for space efficiency. Appending and popping from lists now runs faster due to more efficient code paths and less frequent use of the underlying system `realloc()`. List comprehensions also benefit. `list.extend()` was also optimized and no longer converts its argument into a temporary list before extending the base list. (Contributed by Raymond Hettinger.)

- `list()`, `tuple()`, `map()`, `filter()`, and `zip()` now run several
times faster with non-sequence arguments that supply a

```
__len__()
```

method. (Contributed by Raymond Hettinger.)

- The methods `list.__getitem__()`, `dict.__getitem__()`, and `dict.__contains__()` are now implemented as `method_descriptor` objects rather than `wrapper_descriptor` objects. This form of access doubles their performance and makes them more suitable for use as arguments to functionals:

```
map(mydict.__getitem__, keylist).
```

(Contributed by Raymond Hettinger.)

- Added a new opcode, `LIST_APPEND`, that simplifies the generated bytecode for list comprehensions and speeds them up by about a third. (Contributed by Raymond Hettinger.)

- The peephole bytecode optimizer has been improved to produce shorter, faster bytecode; remarkably, the resulting bytecode is more readable. (Enhanced by Raymond Hettinger.)

- String concatenations in statements of the form `s = s + "abc"` and `s += "abc"` are now performed more efficiently in certain circumstances. This optimization won’t be present in other Python implementations such as Jython, so you shouldn’t rely on it; using the `join()` method of strings is still recommended when you want to efficiently glue a large number of strings together. (Contributed by Armin Rigo.)

The net result of the 2.4 optimizations is that Python 2.4 runs the pystone benchmark around 5% faster than Python 2.3 and 35% faster than Python 2.2. (pystone is not a particularly good benchmark, but it’s the most commonly used measurement of Python’s performance. Your own applications may show greater or smaller benefits from Python 2.4.)
New, Improved, and Deprecated Modules

As usual, Python's standard library received a number of enhancements and bug fixes. Here's a partial list of the most notable changes, sorted alphabetically by module name. Consult the Misc/NEWS file in the source tree for a more complete list of changes, or look through the CVS logs for all the details.

- The `asyncore` module's `loop()` function now has a `count` parameter that lets you perform a limited number of passes through the polling loop. The default is still to loop forever.

- The `base64` module now has more complete RFC 3548 support for Base64, Base32, and Base16 encoding and decoding, including optional case folding and optional alternative alphabets. (Contributed by Barry Warsaw.)

- The `bisect` module now has an underlying C implementation for improved performance. (Contributed by Dmitry Vasiliev.)

- The CJKCodecs collections of East Asian codecs, maintained by Hye-Shik Chang, was integrated into 2.4. The new encodings are:
  - Chinese (PRC): gb2312, gbk, gb18030, big5hkscs, hz
  - Chinese (ROC): big5, cp950
• Some other new encodings were added: HP Roman8, ISO_8859-11, ISO_8859-16, PCTP-154, and TIS-620.

• The UTF-8 and UTF-16 codecs now cope better with receiving partial input. Previously the StreamReader class would try to read more data, making it impossible to resume decoding from the stream. The read() method will now return as much data as it can and future calls will resume decoding where previous ones left off. (Implemented by Walter Dörwald.)

• There is a new collections module for various specialized collection datatypes. Currently it contains just one type, deque, a double-ended queue that supports efficiently adding and removing elements from either end:

```python
>>> from collections import deque
>>> d = deque('ghi')  # make a new deque with three items
>>> d.append('j')  # add a new entry to the right side
>>> d.appendleft('f')  # add a new entry to the left side
>>> d  # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])
>>> d.pop()  # return and remove the rightmost item
'j'
>>> d.popleft()  # return and remove the leftmost item
'f'
>>> list(d)  # list the contents of the deque
['g', 'h', 'i']
>>> 'h' in d  # search the deque
True
```

Several modules, such as the Queue and threading modules, now take advantage of collections.deque for improved performance. (Contributed by Raymond Hettinger.)

• The ConfigParser classes have been enhanced slightly. The read() method now returns a list of the files that were successfully parsed, and the set() method raises TypeError if
passed a value argument that isn’t a string. (Contributed by John Belmonte and David Goodger.)

- The curses module now supports the ncurses extension use_default_colors(). On platforms where the terminal supports transparency, this makes it possible to use a transparent background. (Contributed by Jörg Lehmann.)

- The difflib module now includes an HtmlDiff class that creates an HTML table showing a side by side comparison of two versions of a text. (Contributed by Dan Gass.)

- The email package was updated to version 3.0, which dropped various deprecated APIs and removes support for Python versions earlier than 2.3. The 3.0 version of the package uses a new incremental parser for MIME messages, available in the email.FeedParser module. The new parser doesn’t require reading the entire message into memory, and doesn’t raise exceptions if a message is malformed; instead it records any problems in the defect attribute of the message. (Developed by Anthony Baxter, Barry Warsaw, Thomas Wouters, and others.)

- The heapq module has been converted to C. The resulting tenfold improvement in speed makes the module suitable for handling high volumes of data. In addition, the module has two new functions nlargest() and nsmallest() that use heaps to find the N largest or smallest values in a dataset without the expense of a full sort. (Contributed by Raymond Hettinger.)

- The httplib module now contains constants for HTTP status codes defined in various HTTP-related RFC documents. Constants have names such as OK, CREATED, CONTINUE, and MOVED_PERMANENTLY; use pydoc to get a full list. (Contributed by Andrew Eland.)
• The `imaplib` module now supports IMAP's THREAD command (contributed by Yves Dionne) and new `deleteacl()` and `myrights()` methods (contributed by Arnaud Mazin).

• The `itertools` module gained a `groupby(iterable[, *func*])()` function. `iterable` is something that can be iterated over to return a stream of elements, and the optional `func` parameter is a function that takes an element and returns a key value; if omitted, the key is simply the element itself. `groupby()` then groups the elements into subsequences which have matching values of the key, and returns a series of 2-tuples containing the key value and an iterator over the subsequence.

Here's an example to make this clearer. The `key` function simply returns whether a number is even or odd, so the result of `groupby()` is to return consecutive runs of odd or even numbers.

```python
>>> import itertools
>>> L = [2, 4, 6, 7, 8, 9, 11, 12, 14]
>>> for key_val, it in itertools.groupby(L, lambda x: x % 2):
...     print(key_val, list(it))
... 0 [2, 4, 6]
1 [7]
0 [8]
1 [9, 11]
0 [12, 14]
```

`groupby()` is typically used with sorted input. The logic for `groupby()` is similar to the Unix `uniq` filter which makes it handy for eliminating, counting, or identifying duplicate elements:

```python
>>> word = 'abracadabra'
>>> letters = sorted(word)  # Turn string into a sorted list
>>> letters
['a', 'a', 'a', 'a', 'b', 'b', 'c', 'd', 'r', 'r']
```
```python
>>> for k, g in itertools.groupby(letters):
    ...   print k, list(g)
...
a ['a', 'a', 'a', 'a', 'a']
b ['b', 'b']
c ['c']
d ['d']
r ['r', 'r']
>>> # List unique letters
>>> [k for k, g in groupby(letters)]
['a', 'b', 'c', 'd', 'r']
>>> # Count letter occurrences
>>> [(k, len(list(g))) for k, g in groupby(letters)]
[('a', 5), ('b', 2), ('c', 1), ('d', 1), ('r', 2)]
```

(Contributed by Hye-Shik Chang.)

- **itertools** also gained a function named `tee(iterator, N)()` that returns $N$ independent iterators that replicate `iterator`. If $N$ is omitted, the default is 2.

```python
>>> L = [1,2,3]
>>> i1, i2 = itertools.tee(L)
>>> i1, i2
(<itertools.tee object at 0x402c2080>, <itertools.tee object at 0x402c2090>)
>>> list(i1)                    # Run the first iterator to exhaustion
[1, 2, 3]
>>> list(i2)                    # Run the second iterator to exhaustion
[1, 2, 3]
```

Note that `tee()` has to keep copies of the values returned by the iterator; in the worst case, it may need to keep all of them. This should therefore be used carefully if the leading iterator can run far ahead of the trailing iterator in a long stream of inputs. If the separation is large, then you might as well use `list()` instead. When the iterators track closely with one another, `tee()` is ideal. Possible applications include bookmarking, windowing, or lookahead iterators. (Contributed by Raymond Hettinger.)
A number of functions were added to the `locale` module, such as `bind_textdomain_codeset()` to specify a particular encoding and a family of `gettext()` functions that return messages in the chosen encoding. (Contributed by Gustavo Niemeyer.)

Some keyword arguments were added to the `logging` package's `basicConfig()` function to simplify log configuration. The default behavior is to log messages to standard error, but various keyword arguments can be specified to log to a particular file, change the logging format, or set the logging level. For example:

```python
import logging
logging.basicConfig(filename='/var/log/application.log',
    level=0,  # Log all messages
    format='%(levelname):%(process):%(thread):%(message)'
)
```

Other additions to the `logging` package include a `log(level, msg)()` convenience method, as well as a `TimedRotatingFileHandler` class that rotates its log files at a timed interval. The module already had `RotatingFileHandler`, which rotated logs once the file exceeded a certain size. Both classes derive from a new `BaseRotatingHandler` class that can be used to implement other rotating handlers.

(Changes implemented by Vinay Sajip.)

The `marshal` module now shares interned strings on unpacking a data structure. This may shrink the size of certain pickle strings, but the primary effect is to make `.pyc` files significantly smaller. (Contributed by Martin von Löwis.)

The `nntplib` module's `NNTP` class gained `description()` and `descriptions()` methods to retrieve newsgroup descriptions for a single group or for a range of groups. (Contributed by Jürgen A. Erhard.)
Two new functions were added to the `operator` module, `attrgetter(attr)`() and `itemgetter(index)`(). Both functions return callables that take a single argument and return the corresponding attribute or item; these callables make excellent data extractors when used with `map()` or `sorted()`. For example:

```python
>>> L = [('c', 2), ('d', 1), ('a', 4), ('b', 3)]
>>> map(operator.itemgetter(0), L)
['c', 'd', 'a', 'b']
>>> map(operator.itemgetter(1), L)
[2, 1, 4, 3]
>>> sorted(L, key=operator.itemgetter(1))  # Sort list by second tuple item
[('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

(Contributed by Raymond Hettinger.)

The `optparse` module was updated in various ways. The module now passes its messages through `gettext.gettext()`, making it possible to internationalize Optik's help and error messages. Help messages for options can now include the string `'%default'`, which will be replaced by the option's default value. (Contributed by Greg Ward.)

The long-term plan is to deprecate the `rfc822` module in some future Python release in favor of the `email` package. To this end, the `email.Utils.formatdate()` function has been changed to make it usable as a replacement for `rfc822.formatdate()`. You may want to write new e-mail processing code with this in mind. (Change implemented by Anthony Baxter.)

A new `urandom(n)`() function was added to the `os` module, returning a string containing `n` bytes of random data. This function provides access to platform-specific sources of randomness such as `/dev/urandom` on Linux or the Windows
CryptoAPI. (Contributed by Trevor Perrin.)

- Another new function: `os.path.lexists(path)()` returns true if the file specified by `path` exists, whether or not it’s a symbolic link. This differs from the existing `os.path.exists(path)()` function, which returns false if `path` is a symlink that points to a destination that doesn’t exist. (Contributed by Beni Cherniavsky.)

- A new `getsid()` function was added to the `posix` module that underlies the `os` module. (Contributed by J. Raynor.)

- The `poplib` module now supports POP over SSL. (Contributed by Hector Urtubia.)

- The `profile` module can now profile C extension functions. (Contributed by Nick Bastin.)

- The `random` module has a new method called `getrandbits(N)()` that returns a long integer `N` bits in length. The existing `randrange()` method now uses `getrandbits()` where appropriate, making generation of arbitrarily large random numbers more efficient. (Contributed by Raymond Hettinger.)

- The regular expression language accepted by the `re` module was extended with simple conditional expressions, written as `(<? (group)A|B)`. `group` is either a numeric group ID or a group name defined with `(?P<group>...)` earlier in the expression. If the specified group matched, the regular expression pattern `A` will be tested against the string; if the group didn’t match, the pattern `B` will be used instead. (Contributed by Gustavo Niemeyer.)

- The `re` module is also no longer recursive, thanks to a massive
amount of work by Gustavo Niemeyer. In a recursive regular expression engine, certain patterns result in a large amount of C stack space being consumed, and it was possible to overflow the stack. For example, if you matched a 30000-byte string of a characters against the expression (a|b)+, one stack frame was consumed per character. Python 2.3 tried to check for stack overflow and raise a `RuntimeError` exception, but certain patterns could sidestep the checking and if you were unlucky Python could segfault. Python 2.4’s regular expression engine can match this pattern without problems.

- The `signal` module now performs tighter error-checking on the parameters to the `signal.signal()` function. For example, you can’t set a handler on the `SIGKILL` signal; previous versions of Python would quietly accept this, but 2.4 will raise a `RuntimeError` exception.

- Two new functions were added to the `socket` module. `socketpair()` returns a pair of connected sockets and `getservbyport(port)`() looks up the service name for a given port number. (Contributed by Dave Cole and Barry Warsaw.)

- The `sys.exitfunc()` function has been deprecated. Code should be using the existing `atexit` module, which correctly handles calling multiple exit functions. Eventually `sys.exitfunc()` will become a purely internal interface, accessed only by `atexit`.

- The `tarfile` module now generates GNU-format tar files by default. (Contributed by Lars Gustaebel.)

- The `threading` module now has an elegantly simple way to support thread-local data. The module contains a `local` class whose attribute values are local to different threads.
import threading

data = threading.local()
data.number = 42
data.url = ('www.python.org', 80)

Other threads can assign and retrieve their own values for the number and url attributes. You can subclass local to initialize attributes or to add methods. (Contributed by Jim Fulton.)

- The timeit module now automatically disables periodic garbage collection during the timing loop. This change makes consecutive timings more comparable. (Contributed by Raymond Hettinger.)

- The weakref module now supports a wider variety of objects including Python functions, class instances, sets, frozensets, deques, arrays, files, sockets, and regular expression pattern objects. (Contributed by Raymond Hettinger.)

- The xmlrpclib module now supports a multi-call extension for transmitting multiple XML-RPC calls in a single HTTP operation. (Contributed by Brian Quinlan.)

- The mpz, rotor, and xreadlines modules have been removed.

cookieilib

The cookielib library supports client-side handling for HTTP cookies, mirroring the Cookie module’s server-side cookie support. Cookies are stored in cookie jars; the library transparently stores cookies offered by the web server in the cookie jar, and fetches the cookie from the jar when connecting to the server. As in web browsers, policy objects control whether cookies are accepted or not.
In order to store cookies across sessions, two implementations of cookie jars are provided: one that stores cookies in the Netscape format so applications can use the Mozilla or Lynx cookie files, and one that stores cookies in the same format as the Perl libwww library.

`urllib2` has been changed to interact with `cookielib`: `HTTPCookieProcessor` manages a cookie jar that is used when accessing URLs.

This module was contributed by John J. Lee.

---

### doctest

The `doctest` module underwent considerable refactoring thanks to Edward Loper and Tim Peters. Testing can still be as simple as running `doctest.testmod()`, but the refactorings allow customizing the module’s operation in various ways.

The new `DocTestFinder` class extracts the tests from a given object’s docstrings:

```python
def f(x, y):
    """>>> f(2,2) 4
    >>> f(3,2) 6
    """
    return x*y

finder = doctest.DocTestFinder()

# Get list of DocTest instances
tests = finder.find(f)
```

The new `DocTestRunner` class then runs individual tests and can produce a summary of the results:
runner = doctest.DocTestRunner()
for t in tests:
    tried, failed = runner.run(t)
runner.summarize(verbos=1)

The above example produces the following output:

```python
1 items passed all tests:
  2 tests in f
2 tests in 1 items.
2 passed and 0 failed.
Test passed.
```

```
DocTestRunner uses an instance of the OutputChecker class to compare the expected output with the actual output. This class takes a number of different flags that customize its behaviour; ambitious users can also write a completely new subclass of OutputChecker.

The default output checker provides a number of handy features. For example, with the doctest.ELLIPSIS option flag, an ellipsis (...) in the expected output matches any substring, making it easier to accommodate outputs that vary in minor ways:
```
def o(n):
    
    o(1)
<_main__.C instance at 0x...>
    >>>

Another special string, <BLANKLINE>, matches a blank line:
```
def p(n):
    
    p(1)
<BLANKLINE>
    >>>

Another new capability is producing a diff-style display of the output
by specifying the `doctest.REPORT_UDIFF` (unified diffs),
`doctest.REPORT_CDIFF` (context diffs), or `doctest.REPORT_NDIFF` (delta-style) option flags. For example:

```python
def g(n):
    
    """>>> g(4)

here
is
a
lengthy

L = 'here is a rather lengthy list of words'.split()
for word in L[:n]:
    print word
```

Running the above function's tests with `doctest.REPORT_UDIFF` specified, you get the following output:

```
**********************************************************************
File "t.py", line 15, in g
Failed example:
g(4)
Differences (unified diff with -expected +actual):
  @@ -2,3 +2,3 @@
is
a
-lengthy
+rather
**********************************************************************
```
Build and C API Changes

Some of the changes to Python's build process and to the C API are:

- Three new convenience macros were added for common return values from extension functions: `Py_RETURN_NONE`, `Py_RETURN_TRUE`, and `Py_RETURN_FALSE`. (Contributed by Brett Cannon.)
- Another new macro, `Py_CLEAR(obj)`, decreases the reference count of `obj` and sets `obj` to the null pointer. (Contributed by Jim Fulton.)
- A new function, `PyTuple_Pack(N, obj1, obj2, ..., objN)()`, constructs tuples from a variable length argument list of Python objects. (Contributed by Raymond Hettinger.)
- A new function, `PyDict_Contains(d, k)()`, implements fast dictionary lookups without masking exceptions raised during the look-up process. (Contributed by Raymond Hettinger.)
- The `Py_IS_NAN(X)` macro returns 1 if its float or double argument `X` is a NaN. (Contributed by Tim Peters.)
- C code can avoid unnecessary locking by using the new `PyEval_ThreadsInitialized()` function to tell if any thread operations have been performed. If this function returns false, no lock operations are needed. (Contributed by Nick Coghlan.)
- A new function, `PyArg_VaParseTupleAndKeywords()`, is the same as `PyArg_ParseTupleAndKeywords()` but takes a `va_list` instead of a number of arguments. (Contributed by Greg Chapman.)
- A new method flag, `METH_COEXISTS`, allows a function defined in slots to co-exist with a `PyCFunction` having the same name. This can halve the access time for a method such as `set.__contains__()`. (Contributed by Raymond Hettinger.)
- Python can now be built with additional profiling for the interpreter itself, intended as an aid to people developing the
Python core. Providing `--enable-profiling` to the `configure` script will let you profile the interpreter with `gprof`, and providing the `--with-tsc` switch enables profiling using the Pentium’s Time-Stamp-Counter register. Note that the `--with-tsc` switch is slightly misnamed, because the profiling feature also works on the PowerPC platform, though that processor architecture doesn’t call that register “the TSC register”. (Contributed by Jeremy Hylton.)

- The `tracebackobject` type has been renamed to `PyTracebackObject`.

## Port-Specific Changes

- The Windows port now builds under MSVC++ 7.1 as well as version 6. (Contributed by Martin von Löwis.)
Porting to Python 2.4

This section lists previously described changes that may require changes to your code:

- Left shifts and hexadecimal/octal constants that are too large no longer trigger a `FutureWarning` and return a value limited to 32 or 64 bits; instead they return a long integer.
- Integer operations will no longer trigger an `OverflowWarning`. The `OverflowWarning` warning will disappear in Python 2.5.
- The `zip()` built-in function and `itertools.izip()` now return an empty list instead of raising a `TypeError` exception if called with no arguments.
- You can no longer compare the `date` and `datetime` instances provided by the `datetime` module. Two instances of different classes will now always be unequal, and relative comparisons (`<, >`) will raise a `TypeError`.
- `dircache.listdir()` now passes exceptions to the caller instead of returning empty lists.
- `LexicalHandler.startDTD()` used to receive the public and system IDs in the wrong order. This has been corrected; applications relying on the wrong order need to be fixed.
- `fcntl.ioctl()` now warns if the `mutate` argument is omitted and relevant.
- The `tarfile` module now generates GNU-format tar files by default.
- Encountering a failure while importing a module no longer leaves a partially-initialized module object in `sys.modules`.
- `None` is now a constant; code that binds a new value to the name `None` is now a syntax error.
- The `signals.signal()` function now raises a `RuntimeError`
exception for certain illegal values; previously these errors would pass silently. For example, you can no longer set a handler on the **SIGKILL** signal.
Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Koray Can, Hye-Shik Chang, Michael Dyck, Raymond Hettinger, Brian Hurt, Hamish Lawson, Fredrik Lundh, Sean Reifschneider, Sadruddin Rejeb.
What’s New in Python 2.3

Author: A.M. Kuchling

This article explains the new features in Python 2.3. Python 2.3 was released on July 29, 2003.

The main themes for Python 2.3 are polishing some of the features added in 2.2, adding various small but useful enhancements to the core language, and expanding the standard library. The new object model introduced in the previous version has benefited from 18 months of bugfixes and from optimization efforts that have improved the performance of new-style classes. A few new built-in functions have been added such as `sum()` and `enumerate()`. The `in` operator can now be used for substring searches (e.g. "ab" in "abc" returns `True`).

Some of the many new library features include Boolean, set, heap, and date/time data types, the ability to import modules from ZIP-format archives, metadata support for the long-awaited Python catalog, an updated version of IDLE, and modules for logging messages, wrapping text, parsing CSV files, processing command-line options, using BerkeleyDB databases... the list of new and enhanced modules is lengthy.

This article doesn’t attempt to provide a complete specification of the new features, but instead provides a convenient overview. For full details, you should refer to the documentation for Python 2.3, such as the Python Library Reference and the Python Reference Manual. If you want to understand the complete implementation and design rationale, refer to the PEP for a particular new feature.
The new \texttt{sets} module contains an implementation of a set datatype. The \texttt{Set} class is for mutable sets, sets that can have members added and removed. The \texttt{ImmutableSet} class is for sets that can't be modified, and instances of \texttt{ImmutableSet} can therefore be used as dictionary keys. Sets are built on top of dictionaries, so the elements within a set must be hashable.

Here's a simple example:

```python
>>> import sets
>>> S = sets.Set([1,2,3])
>>> S
Set([1, 2, 3])
>>> 1 in S
True
>>> 0 in S
False
>>> S.add(5)
>>> S
Set([1, 2, 5])
```

The union and intersection of sets can be computed with the \texttt{union()} and \texttt{intersection()} methods; an alternative notation uses the bitwise operators \& and |. Mutable sets also have in-place versions of these methods, \texttt{union_update()} and \texttt{intersection_update()}.

```python
>>> S1 = sets.Set([1,2,3])
>>> S2 = sets.Set([4,5,6])
>>> S1.union(S2)
Set([1, 2, 3, 4, 5, 6])
>>> S1 | S2  # Alternative notation
Set([1, 2, 3, 4, 5, 6])
>>> S1.intersection(S2)
Set([])
```
It's also possible to take the symmetric difference of two sets. This is the set of all elements in the union that aren't in the intersection. Another way of putting it is that the symmetric difference contains all elements that are in exactly one set. Again, there's an alternative notation (\(^\)), and an in-place version with the ungainly name symmetric_difference_update().

See also:

**PEP 218 - Adding a Built-In Set Object Type**

PEP written by Greg V. Wilson. Implemented by Greg V. Wilson, Alex Martelli, and GvR.
In Python 2.2, generators were added as an optional feature, to be enabled by a `from __future__ import generators` directive. In 2.3 generators no longer need to be specially enabled, and are now always present; this means that `yield` is now always a keyword. The rest of this section is a copy of the description of generators from the “What’s New in Python 2.2” document; if you read it back when Python 2.2 came out, you can skip the rest of this section.

You’re doubtless familiar with how function calls work in Python or C. When you call a function, it gets a private namespace where its local variables are created. When the function reaches a `return` statement, the local variables are destroyed and the resulting value is returned to the caller. A later call to the same function will get a fresh new set of local variables. But, what if the local variables weren’t thrown away on exiting a function? What if you could later resume the function where it left off? This is what generators provide; they can be thought of as resumable functions.

Here’s the simplest example of a generator function:

```python
def generate_ints(N):
    for i in range(N):
        yield i
```

A new keyword, `yield`, was introduced for generators. Any function containing a `yield` statement is a generator function; this is detected by Python’s bytecode compiler which compiles the function specially as a result.

When you call a generator function, it doesn’t return a single value; instead it returns a generator object that supports the iterator protocol. On executing the `yield` statement, the generator outputs
the value of $i$, similar to a return statement. The big difference between yield and a return statement is that on reaching a yield the generator’s state of execution is suspended and local variables are preserved. On the next call to the generator’s .next() method, the function will resume executing immediately after the yield statement. (For complicated reasons, the yield statement isn’t allowed inside the try block of a try...finally statement; read PEP 255 for a full explanation of the interaction between yield and exceptions.)

Here’s a sample usage of the `generate_ints()` generator:

```python
>>> gen = generate_ints(3)
>>> gen
<generator object at 0x8117f90>
>>> gen.next()
0
>>> gen.next()
1
>>> gen.next()
2
>>> gen.next()
Traceback (most recent call last):
  File "stdin", line 1, in ?
  File "stdin", line 2, in generate_ints
StopIteration
```

You could equally write `for i in generate_ints(5), or a,b,c = generate_ints(3).`

Inside a generator function, the return statement can only be used without a value, and signals the end of the procession of values; afterwards the generator cannot return any further values. return with a value, such as `return 5`, is a syntax error inside a generator function. The end of the generator’s results can also be indicated by raising StopIteration manually, or by just letting the flow of execution fall off the bottom of the function.
You could achieve the effect of generators manually by writing your own class and storing all the local variables of the generator as instance variables. For example, returning a list of integers could be done by setting `self.count` to 0, and having the `next()` method increment `self.count` and return it. However, for a moderately complicated generator, writing a corresponding class would be much messier. `Lib/test/test_generators.py` contains a number of more interesting examples. The simplest one implements an in-order traversal of a tree using generators recursively.

```python
# A recursive generator that generates Tree leaves in in-order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x
        yield t.label
        for x in inorder(t.right):
            yield x
```

Two other examples in `Lib/test/test_generators.py` produce solutions for the N-Queens problem (placing $N$ queens on an $N\times N$ chess board so that no queen threatens another) and the Knight’s Tour (a route that takes a knight to every square of an $N\times N$ chessboard without visiting any square twice).

The idea of generators comes from other programming languages, especially Icon ([https://www.cs.arizona.edu/icon/](https://www.cs.arizona.edu/icon/)), where the idea of generators is central. In Icon, every expression and function call behaves like a generator. One example from “An Overview of the Icon Programming Language” at [https://www.cs.arizona.edu/icon/docs/ipd266.htm](https://www.cs.arizona.edu/icon/docs/ipd266.htm) gives an idea of what this looks like:

```icon
sentence := "Store it in the neighboring harbor"
if (i := find("or", sentence)) > 5 then write(i)
```
In Icon the `find()` function returns the indexes at which the substring "or" is found: 3, 23, 33. In the `if` statement, `i` is first assigned a value of 3, but 3 is less than 5, so the comparison fails, and Icon retries it with the second value of 23. 23 is greater than 5, so the comparison now succeeds, and the code prints the value 23 to the screen.

Python doesn’t go nearly as far as Icon in adopting generators as a central concept. Generators are considered part of the core Python language, but learning or using them isn’t compulsory; if they don’t solve any problems that you have, feel free to ignore them. One novel feature of Python’s interface as compared to Icon’s is that a generator’s state is represented as a concrete object (the iterator) that can be passed around to other functions or stored in a data structure.

See also:

**PEP 255 - Simple Generators**
Written by Neil Schemenauer, Tim Peters, Magnus Lie Hetland. Implemented mostly by Neil Schemenauer and Tim Peters, with other fixes from the Python Labs crew.
Python source files can now be declared as being in different character set encodings. Encodings are declared by including a specially formatted comment in the first or second line of the source file. For example, a UTF-8 file can be declared with:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
```

Without such an encoding declaration, the default encoding used is 7-bit ASCII. Executing or importing modules that contain string literals with 8-bit characters and have no encoding declaration will result in a `DeprecationWarning` being signalled by Python 2.3; in 2.4 this will be a syntax error.

The encoding declaration only affects Unicode string literals, which will be converted to Unicode using the specified encoding. Note that Python identifiers are still restricted to ASCII characters, so you can't have variable names that use characters outside of the usual alphanumerics.

**See also:**

**PEP 263 - Defining Python Source Code Encodings**

Written by Marc-André Lemburg and Martin von Löwis; implemented by Suzuki Hisao and Martin von Löwis.
PEP 273: Importing Modules from ZIP Archives

The new `zipimport` module adds support for importing modules from a ZIP-format archive. You don’t need to import the module explicitly; it will be automatically imported if a ZIP archive’s filename is added to `sys.path`. For example:

```
+-------------------+-------------------+-------------------+---------------+
<table>
<thead>
<tr>
<th>Archivename</th>
<th>Length</th>
<th>Date</th>
<th>Time</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>/tmp/example.zip</td>
<td>8467</td>
<td>11-26-02</td>
<td>22:30</td>
<td>jwzthreading.py</td>
</tr>
</tbody>
</table>
+-------------------+--------+----------+--------+---------------+
```

```
>>> import sys
>>> sys.path.insert(0, '/tmp/example.zip')  # Add .zip file to
>>> import jwzthreading
>>> jwzthreading.__file__
'/tmp/example.zip/jwzthreading.py'
```

An entry in `sys.path` can now be the filename of a ZIP archive. The ZIP archive can contain any kind of files, but only files named `*.py`, `*.pyc`, or `*.pyo` can be imported. If an archive only contains `*.py` files, Python will not attempt to modify the archive by adding the corresponding `*.pyc` file, meaning that if a ZIP archive doesn't contain `*.pyc` files, importing may be rather slow.

A path within the archive can also be specified to only import from a subdirectory; for example, the path `/tmp/example.zip/lib/` would only import from the `lib/` subdirectory within the archive.
See also:

**PEP 273 - Import Modules from Zip Archives**
Written by James C. Ahlstrom, who also provided an implementation. Python 2.3 follows the specification in PEP 273, but uses an implementation written by Just van Rossum that uses the import hooks described in PEP 302. See section PEP 302: New Import Hooks for a description of the new import hooks.
On Windows NT, 2000, and XP, the system stores file names as Unicode strings. Traditionally, Python has represented file names as byte strings, which is inadequate because it renders some file names inaccessible.

Python now allows using arbitrary Unicode strings (within the limitations of the file system) for all functions that expect file names, most notably the `open()` built-in function. If a Unicode string is passed to `os.listdir()`, Python now returns a list of Unicode strings. A new function, `os.getcwdu()`, returns the current directory as a Unicode string.

Byte strings still work as file names, and on Windows Python will transparently convert them to Unicode using the mbcs encoding.

Other systems also allow Unicode strings as file names but convert them to byte strings before passing them to the system, which can cause a `UnicodeError` to be raised. Applications can test whether arbitrary Unicode strings are supported as file names by checking `os.path.supports_unicode_filenames`, a Boolean value.

Under MacOS, `os.listdir()` may now return Unicode filenames.

See also:

**PEP 277 - Unicode file name support for Windows NT**

Written by Neil Hodgson; implemented by Neil Hodgson, Martin von Löwis, and Mark Hammond.
PEP 278: Universal Newline Support

The three major operating systems used today are Microsoft Windows, Apple’s Macintosh OS, and the various Unix derivatives. A minor irritation of cross-platform work is that these three platforms all use different characters to mark the ends of lines in text files. Unix uses the linefeed (ASCII character 10), MacOS uses the carriage return (ASCII character 13), and Windows uses a two-character sequence of a carriage return plus a newline.

Python's file objects can now support end of line conventions other than the one followed by the platform on which Python is running. Opening a file with the mode 'u' or 'ru' will open a file for reading in universal newlines mode. All three line ending conventions will be translated to a '
' in the strings returned by the various file methods such as read() and readline().

Universal newline support is also used when importing modules and when executing a file with the execfile() function. This means that Python modules can be shared between all three operating systems without needing to convert the line-endings.

This feature can be disabled when compiling Python by specifying the --without-universal-newlines switch when running Python's configure script.

See also:

PEP 278 - Universal Newline Support
Written and implemented by Jack Jansen.
A new built-in function, `enumerate()`, will make certain loops a bit clearer. `enumerate(thing)`, where `thing` is either an iterator or a sequence, returns an iterator that will return `(0, thing[0]), (1, thing[1]), (2, thing[2])`, and so forth.

A common idiom to change every element of a list looks like this:

```python
for i in range(len(L)):
    item = L[i]
    # ... compute some result based on item ...
    L[i] = result
```

This can be rewritten using `enumerate()` as:

```python
for i, item in enumerate(L):
    # ... compute some result based on item ...
    L[i] = result
```

**See also:**

**PEP 279 - The `enumerate()` built-in function**

Written and implemented by Raymond D. Hettinger.
PEP 282: The logging Package

A standard package for writing logs, `logging`, has been added to Python 2.3. It provides a powerful and flexible mechanism for generating logging output which can then be filtered and processed in various ways. A configuration file written in a standard format can be used to control the logging behavior of a program. Python includes handlers that will write log records to standard error or to a file or socket, send them to the system log, or even e-mail them to a particular address; of course, it's also possible to write your own handler classes.

The `Logger` class is the primary class. Most application code will deal with one or more `Logger` objects, each one used by a particular subsystem of the application. Each `Logger` is identified by a name, and names are organized into a hierarchy using `.` as the component separator. For example, you might have `Logger` instances named `server`, `server.auth` and `server.network`. The latter two instances are below `server` in the hierarchy. This means that if you turn up the verbosity for `server` or direct `server` messages to a different handler, the changes will also apply to records logged to `server.auth` and `server.network`. There's also a root `Logger` that's the parent of all other loggers.

For simple uses, the `logging` package contains some convenience functions that always use the root log:

```python
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning: config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```
This produces the following output:

WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down

In the default configuration, informational and debugging messages are suppressed and the output is sent to standard error. You can enable the display of informational and debugging messages by calling the setLevel() method on the root logger.

Notice the warning() call's use of string formatting operators; all of the functions for logging messages take the arguments (msg, arg1, arg2, ...) and log the string resulting from msg % (arg1, arg2, ...).

There's also an exception() function that records the most recent traceback. Any of the other functions will also record the traceback if you specify a true value for the keyword argument exc_info.

```python
def f():
    try:
        1/0
    except:
        logging.exception('Problem recorded')

f()
```

This produces the following output:

ERROR:root:Problem recorded
Traceback (most recent call last):
  File "t.py", line 6, in f
  1/0
ZeroDivisionError: integer division or modulo by zero

Slightly more advanced programs will use a logger other than the root logger. The getLogger(name)() function is used to get a
particular log, creating it if it doesn't exist yet. `getLogger(None)()` returns the root logger.

```python
log = logging.getLogger('server')
...
log.info('Listening on port %i', port)
...
log.critical('Disk full')
```

Log records are usually propagated up the hierarchy, so a message logged to `server.auth` is also seen by `server` and `root`, but a `Logger` can prevent this by setting its `propagate` attribute to `False`.

There are more classes provided by the `logging` package that can be customized. When a `Logger` instance is told to log a message, it creates a `LogRecord` instance that is sent to any number of different `Handler` instances. Loggers and handlers can also have an attached list of filters, and each filter can cause the `LogRecord` to be ignored or can modify the record before passing it along. When they're finally output, `LogRecord` instances are converted to text by a `Formatter` class. All of these classes can be replaced by your own specially-written classes.

With all of these features the `logging` package should provide enough flexibility for even the most complicated applications. This is only an incomplete overview of its features, so please see the package's reference documentation for all of the details. Reading `PEP 282` will also be helpful.

**See also:**

**PEP 282 - A Logging System**

Written by Vinay Sajip and Trent Mick; implemented by Vinay Sajip.
PEP 285: A Boolean Type

A Boolean type was added to Python 2.3. Two new constants were added to the \_\_builtin\_ module, True and False. (True and False constants were added to the built-ins in Python 2.2.1, but the 2.2.1 versions are simply set to integer values of 1 and 0 and aren't a different type.)

The type object for this new type is named bool; the constructor for it takes any Python value and converts it to True or False.

```python
>>> bool(1)
True
>>> bool(0)
False
>>> bool([])
False
>>> bool((1,))
True
```

Most of the standard library modules and built-in functions have been changed to return Booleans.

```python
>>> obj = []
>>> hasattr(obj, 'append')
True
>>> isinstance(obj, list)
True
>>> isinstance(obj, tuple)
False
```

Python's Booleans were added with the primary goal of making code clearer. For example, if you're reading a function and encounter the statement return 1, you might wonder whether the 1 represents a Boolean truth value, an index, or a coefficient that multiplies some other quantity. If the statement is return True, however, the meaning of the return value is quite clear.
Python's Booleans were not added for the sake of strict type-checking. A very strict language such as Pascal would also prevent you performing arithmetic with Booleans, and would require that the expression in an `if` statement always evaluate to a Boolean result. Python is not this strict and never will be, as PEP 285 explicitly says. This means you can still use any expression in an `if` statement, even ones that evaluate to a list or tuple or some random object. The Boolean type is a subclass of the `int` class so that arithmetic using a Boolean still works.

```python
>>> True + 1
2
>>> False + 1
1
>>> False * 75
0
>>> True * 75
75
```

To sum up `True` and `False` in a sentence: they’re alternative ways to spell the integer values 1 and 0, with the single difference that `str()` and `repr()` return the strings `'True'` and `'False'` instead of `'1'` and `'0'`.

See also:

**PEP 285 - Adding a bool type**

Written and implemented by GvR.
When encoding a Unicode string into a byte string, unencodable characters may be encountered. So far, Python has allowed specifying the error processing as either “strict” (raising `UnicodeError`), “ignore” (skipping the character), or “replace” (using a question mark in the output string), with “strict” being the default behavior. It may be desirable to specify alternative processing of such errors, such as inserting an XML character reference or HTML entity reference into the converted string.

Python now has a flexible framework to add different processing strategies. New error handlers can be added with `codecs.register_error()`, and `codecs` then can access the error handler with `codecs.lookup_error()`. An equivalent C API has been added for codecs written in C. The error handler gets the necessary state information such as the string being converted, the position in the string where the error was detected, and the target encoding. The handler can then either raise an exception or return a replacement string.

Two additional error handlers have been implemented using this framework: “backslashreplace” uses Python backslash quoting to represent unencodable characters and “xmlcharrefreplace” emits XML character references.

See also:

**PEP 293 - Codec Error Handling Callbacks**

Written and implemented by Walter Dörwald.
Support for the long-requested Python catalog makes its first appearance in 2.3.

The heart of the catalog is the new Distutils register command. Running `python setup.py register` will collect the metadata describing a package, such as its name, version, maintainer, description, &c., and send it to a central catalog server. The resulting catalog is available from https://pypi.python.org/pypi.

To make the catalog a bit more useful, a new optional classifiers keyword argument has been added to the Distutils setup() function. A list of Trove-style strings can be supplied to help classify the software.

Here’s an example setup.py with classifiers, written to be compatible with older versions of the Distutils:

```python
from distutils import core
kw = {'name': 'Quixote',
      'version': '0.5.1',
      'description': 'A highly Pythonic Web application framework
# ...
}

if (hasattr(core, 'setup_keywords') and
    'classifiers' in core.setup_keywords):
    kw['classifiers'] = [
        'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
        'Environment :: No Input/Output (Daemon)',
        'Intended Audience :: Developers'],

core.setup(**kw)
```
The full list of classifiers can be obtained by running `python setup.py register --list-classifiers`.

See also:

**PEP 301 - Package Index and Metadata for Distutils**
Written and implemented by Richard Jones.
PEP 302: New Import Hooks

While it’s been possible to write custom import hooks ever since the `ihooks` module was introduced in Python 1.3, no one has ever been really happy with it because writing new import hooks is difficult and messy. There have been various proposed alternatives such as the `imputil` and `iu` modules, but none of them has ever gained much acceptance, and none of them were easily usable from C code.

PEP 302 borrows ideas from its predecessors, especially from Gordon McMillan’s `iu` module. Three new items are added to the `sys` module:

- `sys.path_hooks` is a list of callable objects; most often they’ll be classes. Each callable takes a string containing a path and either returns an importer object that will handle imports from this path or raises an `ImportError` exception if it can’t handle this path.
- `sys.path_importer_cache` caches importer objects for each path, so `sys.path_hooks` will only need to be traversed once for each path.
- `sys.meta_path` is a list of importer objects that will be traversed before `sys.path` is checked. This list is initially empty, but user code can add objects to it. Additional built-in and frozen modules can be imported by an object added to this list.

Importer objects must have a single method, `find_module(fullname, path=None)`(). ` fullname` will be a module or package name, e.g. `string` or `distutils.core`. `find_module()` must return a loader object that has a single method, `load_module(fullname)`(), that creates and returns the corresponding module object.
Pseudo-code for Python’s new import logic, therefore, looks something like this (simplified a bit; see [PEP 302](https://www.python.org/dev/peps/pep-0302) for the full details):

```python
for mp in sys.meta_path:
    loader = mp(fullname)
    if loader is not None:
        <module> = loader.load_module(fullname)

for path in sys.path:
    for hook in sys.path_hooks:
        try:
            importer = hook(path)
        except ImportError:
            # ImportError, so try the other path hooks
            pass
        else:
            loader = importer.find_module(fullname)
            <module> = loader.load_module(fullname)

# Not found!
raise ImportError
```

See also:

- **PEP 302 - New Import Hooks**
  
  Written by Just van Rossum and Paul Moore. Implemented by Just van Rossum.
PEP 305: Comma-separated Files

Comma-separated files are a format frequently used for exporting data from databases and spreadsheets. Python 2.3 adds a parser for comma-separated files.

Comma-separated format is deceptively simple at first glance:

```
Costs,150,200,3.95
```

Read a line and call `line.split(',')`: what could be simpler? But toss in string data that can contain commas, and things get more complicated:

```
"Costs",150,200,3.95,"Includes taxes, shipping, and sundry item
```

A big ugly regular expression can parse this, but using the new `csv` package is much simpler:

```
import csv

input = open('datafile', 'rb')
reader = csv.reader(input)
for line in reader:
    print line
```

The `reader()` function takes a number of different options. The field separator isn’t limited to the comma and can be changed to any character, and so can the quoting and line-ending characters.

Different dialects of comma-separated files can be defined and registered; currently there are two dialects, both used by Microsoft Excel. A separate `csv.writer` class will generate comma-separated files from a succession of tuples or lists, quoting strings that contain
the delimiter.

See also:

**PEP 305 - CSV File API**

Written and implemented by Kevin Altis, Dave Cole, Andrew McNamara, Skip Montanaro, Cliff Wells.
PEP 307: Pickle Enhancements

The pickle and cPickle modules received some attention during the 2.3 development cycle. In 2.2, new-style classes could be pickled without difficulty, but they weren’t pickled very compactly; PEP 307 quotes a trivial example where a new-style class results in a pickled string three times longer than that for a classic class.

The solution was to invent a new pickle protocol. The pickle.dumps() function has supported a text-or-binary flag for a long time. In 2.3, this flag is redefined from a Boolean to an integer: 0 is the old text-mode pickle format, 1 is the old binary format, and now 2 is a new 2.3-specific format. A new constant, pickle.HIGHEST_PROTOCOL, can be used to select the fanciest protocol available.

Unpickling is no longer considered a safe operation. 2.2’s pickle provided hooks for trying to prevent unsafe classes from being unpickled (specifically, a __safe_for_unpickling__ attribute), but none of this code was ever audited and therefore it’s all been ripped out in 2.3. You should not unpickle untrusted data in any version of Python.

To reduce the pickling overhead for new-style classes, a new interface for customizing pickling was added using three special methods: __getstate__(), __setstate__(), and __getnewargs__(). Consult PEP 307 for the full semantics of these methods.

As a way to compress pickles yet further, it’s now possible to use integer codes instead of long strings to identify pickled classes. The Python Software Foundation will maintain a list of standardized codes; there’s also a range of codes for private use. Currently no codes have been specified.
See also:

**PEP 307 - Extensions to the pickle protocol**

Written and implemented by Guido van Rossum and Tim Peters.
Extended Slices

Ever since Python 1.4, the slicing syntax has supported an optional third “step” or “stride” argument. For example, these are all legal Python syntax: \texttt{L[1:10:2]}, \texttt{L[::1]}\texttt{, L[::1]}\texttt{L[::1]}. This was added to Python at the request of the developers of Numerical Python, which uses the third argument extensively. However, Python's built-in list, tuple, and string sequence types have never supported this feature, raising a \texttt{TypeError} if you tried it. Michael Hudson contributed a patch to fix this shortcoming.

For example, you can now easily extract the elements of a list that have even indexes:

\begin{verbatim}
>>> L = range(10)
>>> L[::2]
[0, 2, 4, 6, 8]
\end{verbatim}

Negative values also work to make a copy of the same list in reverse order:

\begin{verbatim}
>>> L[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
\end{verbatim}

This also works for tuples, arrays, and strings:

\begin{verbatim}
>>> s='abcd'
>>> s[::2]
'ac'
>>> s[::1]
'dcba'
\end{verbatim}

If you have a mutable sequence such as a list or an array you can assign to or delete an extended slice, but there are some differences between assignment to extended and regular slices. Assignment to a regular slice can be used to change the length of the sequence:
Extended slices aren’t this flexible. When assigning to an extended slice, the list on the right hand side of the statement must contain the same number of items as the slice it is replacing:

```python
>>> a = range(3)
>>> a
[0, 1, 2]
>>> a[1:3] = [4, 5, 6]
>>> a
[0, 4, 5, 6]
```

Deletion is more straightforward:

```python
>>> a = range(4)
>>> a
[0, 1, 2, 3]
>>> a[::2]
[0, 2]
>>> a[::2] = [0, -1]
>>> a
[0, 1, -1, 3]
>>> a[::2] = [0, 1, 2]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: attempt to assign sequence of size 3 to extended slice of size 2
```

One can also now pass slice objects to the `__getitem__()` methods of the built-in sequences:

```python
>>> range(10).__getitem__(slice(0, 5, 2))
[0, 2, 4]
```
Or use slice objects directly in subscripts:

```python
range(10)[slice(0, 5, 2)]
[0, 2, 4]
```

To simplify implementing sequences that support extended slicing, slice objects now have a method `indices(length)()` which, given the length of a sequence, returns a `(start, stop, step)` tuple that can be passed directly to `range().indices()` handles omitted and out-of-bounds indices in a manner consistent with regular slices (and this innocuous phrase hides a welter of confusing details!). The method is intended to be used like this:

```python
class FakeSeq:
    ...
    def calc_item(self, i):
        ...
    def __getitem__(self, item):
        if isinstance(item, slice):
            indices = item.indices(len(self))
            return FakeSeq([self.calc_item(i) for i in range(*indices)
        else:
            return self.calc_item(i)
```

From this example you can also see that the built-in `slice` object is now the type object for the slice type, and is no longer a function. This is consistent with Python 2.2, where `int`, `str`, etc., underwent the same change.
Other Language Changes

Here are all of the changes that Python 2.3 makes to the core Python language.

- The `yield` statement is now always a keyword, as described in section PEP 255: Simple Generators of this document.

- A new built-in function `enumerate()` was added, as described in section PEP 279: `enumerate()` of this document.

- Two new constants, `True` and `False` were added along with the built-in `bool` type, as described in section PEP 285: A Boolean Type of this document.

- The `int()` type constructor will now return a long integer instead of raising an `OverflowError` when a string or floating-point number is too large to fit into an integer. This can lead to the paradoxical result that `isinstance(int(expression), int)` is false, but that seems unlikely to cause problems in practice.

- Built-in types now support the extended slicing syntax, as described in section Extended Slices of this document.

- A new built-in function, `sum(iterable, start=0)()`, adds up the numeric items in the iterable object and returns their sum. `sum()` only accepts numbers, meaning that you can’t use it to concatenate a bunch of strings. (Contributed by Alex Martelli.)

- `list.insert(pos, value)` used to insert `value` at the front of the list when `pos` was negative. The behaviour has now been changed to be consistent with slice indexing, so when `pos` is -1 the value will be inserted before the last element, and so forth.
• `list.index(value)`, which searches for value within the list and returns its index, now takes optional `start` and `stop` arguments to limit the search to only part of the list.

• Dictionaries have a new method, `pop(key[, *default*])()`, that returns the value corresponding to key and removes that key/value pair from the dictionary. If the requested key isn’t present in the dictionary, `default` is returned if it’s specified and `KeyError` raised if it isn’t.

```python
>>> d = {1:2}
>>> d
{1: 2}
>>> d.pop(4)
Traceback (most recent call last):
  File "stdin", line 1, in ?
KeyError: 4
>>> d.pop(1)
2
>>> d.pop(1)
Traceback (most recent call last):
  File "stdin", line 1, in ?
KeyError: 'pop(): dictionary is empty'
>>> d
{}
```

There’s also a new class method, `dict.fromkeys(iterable, value)()`), that creates a dictionary with keys taken from the supplied iterator `iterable` and all values set to `value`, defaulting to `None`.

(Patches contributed by Raymond Hettinger.)

Also, the `dict()` constructor now accepts keyword arguments to simplify creating small dictionaries:

```python
>>> dict(red=1, blue=2, green=3, black=4)
{'blue': 2, 'black': 4, 'green': 3, 'red': 1}
```
(Contributed by Just van Rossum.)

- The `assert` statement no longer checks the `__debug__` flag, so you can no longer disable assertions by assigning to `__debug__`. Running Python with the `-O` switch will still generate code that doesn’t execute any assertions.

- Most type objects are now callable, so you can use them to create new objects such as functions, classes, and modules. (This means that the `new` module can be deprecated in a future Python version, because you can now use the type objects available in the `types` module.) For example, you can create a new module object with the following code:

  ```python
  >>> import types
  >>> m = types.ModuleType('abc', 'docstring')
  >>> m
  <module 'abc' (built-in)>
  >>> m.__doc__
  'docstring'
  ```

- A new warning, `PendingDeprecationWarning` was added to indicate features which are in the process of being deprecated. The warning will *not* be printed by default. To check for use of features that will be deprecated in the future, supply `-Walways::PendingDeprecationWarning::` on the command line or use `warnings.filterwarnings()`.

- The process of deprecating string-based exceptions, as in `raise "Error occurred"`, has begun. Raising a string will now trigger `PendingDeprecationWarning`.

- Using `None` as a variable name will now result in a `SyntaxWarning` warning. In a future version of Python, `None` may finally become a keyword.
• The `xreadlines()` method of file objects, introduced in Python 2.1, is no longer necessary because files now behave as their own iterator. `xreadlines()` was originally introduced as a faster way to loop over all the lines in a file, but now you can simply write `for line in file_obj`. File objects also have a new read-only `encoding` attribute that gives the encoding used by the file; Unicode strings written to the file will be automatically converted to bytes using the given encoding.

• The method resolution order used by new-style classes has changed, though you'll only notice the difference if you have a really complicated inheritance hierarchy. Classic classes are unaffected by this change. Python 2.2 originally used a topological sort of a class's ancestors, but 2.3 now uses the C3 algorithm as described in the paper “A Monotonic Superclass Linearization for Dylan”. To understand the motivation for this change, read Michele Simionato's article “Python 2.3 Method Resolution Order”, or read the thread on python-dev starting with the message at https://mail.python.org/pipermail/python-dev/2002-October/029035.html. Samuele Pedroni first pointed out the problem and also implemented the fix by coding the C3 algorithm.

• Python runs multithreaded programs by switching between threads after executing N bytecodes. The default value for N has been increased from 10 to 100 bytecodes, speeding up single-threaded applications by reducing the switching overhead. Some multithreaded applications may suffer slower response time, but that's easily fixed by setting the limit back to a lower number using `sys.setcheckinterval(N)()`. The limit can be retrieved with the new `sys.getcheckinterval()` function.

• One minor but far-reaching change is that the names of extension types defined by the modules included with Python
now contain the module and a '.' in front of the type name. For example, in Python 2.2, if you created a socket and printed its __class__, you’d get this output:

```python
>>> s = socket.socket()
>>> s.__class__
<type 'socket'>
```

In 2.3, you get this:

```python
>>> s.__class__
<type '_socket.socket'>
```

- One of the noted incompatibilities between old- and new-style classes has been removed: you can now assign to the __name__ and __bases__ attributes of new-style classes. There are some restrictions on what can be assigned to __bases__ along the lines of those relating to assigning to an instance’s __class__ attribute.

**String Changes**

- The `in` operator now works differently for strings. Previously, when evaluating `X in Y` where `X` and `Y` are strings, `X` could only be a single character. That’s now changed; `X` can be a string of any length, and `X in Y` will return `True` if `X` is a substring of `Y`. If `X` is the empty string, the result is always `True`.

```python
>>> 'ab' in 'abcd'
True
>>> 'ad' in 'abcd'
False
>>> '' in 'abcd'
True
```

Note that this doesn’t tell you where the substring starts; if you
need that information, use the `find()` string method.

- The `strip()`, `lstrip()`, and `rstrip()` string methods now have an optional argument for specifying the characters to strip. The default is still to remove all whitespace characters:

```python
>>> ' abc '.strip()
'abc'
>>> '<<<<abc>>>>'.strip('<>')
'abc'
>>> '<<<<abc>>>>
'.strip('<>')
'abc<<<<'
>>> u'\u4000\u4001abc\u4000'.strip(u'\u4000')
u'\u4001abc'
```  

(Suggested by Simon Brunning and implemented by Walter Dörwald.)

- The `startswith()` and `endswith()` string methods now accept negative numbers for the `start` and `end` parameters.

- Another new string method is `zfill()`, originally a function in the `string` module. `zfill()` pads a numeric string with zeros on the left until it's the specified width. Note that the `%` operator is still more flexible and powerful than `zfill()`.

```python
>>> '45'.zfill(4)
'0045'
>>> '12345'.zfill(4)
'12345'
>>> 'goofy'.zfill(6)
'0goofy'
```  

(Contributed by Walter Dörwald.)

- A new type object, `basestring`, has been added. Both 8-bit strings and Unicode strings inherit from this type, so
isinstance(obj, basestring) will return True for either kind of string. It's a completely abstract type, so you can't create basestring instances.

- Interned strings are no longer immortal and will now be garbage-collected in the usual way when the only reference to them is from the internal dictionary of interned strings. (Implemented by Oren Tirosh.)

**Optimizations**

- The creation of new-style class instances has been made much faster; they're now faster than classic classes!
- The sort() method of list objects has been extensively rewritten by Tim Peters, and the implementation is significantly faster.
- Multiplication of large long integers is now much faster thanks to an implementation of Karatsuba multiplication, an algorithm that scales better than the O(n^2) required for the grade-school multiplication algorithm. (Original patch by Christopher A. Craig, and significantly reworked by Tim Peters.)
- The SET_LINENO opcode is now gone. This may provide a small speed increase, depending on your compiler's idiosyncrasies. See section Other Changes and Fixes for a longer explanation. (Removed by Michael Hudson.)
- xrange() objects now have their own iterator, making for i in xrange(n) slightly faster than for i in range(n). (Patch by Raymond Hettinger.)
- A number of small rearrangements have been made in various hotspots to improve performance, such as inlining a function or removing some code. (Implemented mostly by GvR, but lots of people have contributed single changes.)

The net result of the 2.3 optimizations is that Python 2.3 runs the pystone benchmark around 25% faster than Python 2.2.
New, Improved, and Deprecated Modules

As usual, Python's standard library received a number of enhancements and bug fixes. Here's a partial list of the most notable changes, sorted alphabetically by module name. Consult the Misc/NEWS file in the source tree for a more complete list of changes, or look through the CVS logs for all the details.

- The array module now supports arrays of Unicode characters using the 'u' format character. Arrays also now support using the += assignment operator to add another array's contents, and the *= assignment operator to repeat an array. (Contributed by Jason Orendorff.)

- The bsddb module has been replaced by version 4.1.6 of the PyBSDDDB package, providing a more complete interface to the transactional features of the BerkeleyDB library.

The old version of the module has been renamed to bsddb185 and is no longer built automatically; you'll have to edit Modules/Setup to enable it. Note that the new bsddb package is intended to be compatible with the old module, so be sure to file bugs if you discover any incompatibilities. When upgrading to Python 2.3, if the new interpreter is compiled with a new version of the underlying BerkeleyDB library, you will almost certainly have to convert your database files to the new version. You can do this fairly easily with the new scripts db2pickle.py and pickle2db.py which you will find in the distribution's Tools/scripts directory. If you've already been using the PyBSDDDB package and importing it as bsddb3, you will have to change your import statements to import it as bsddb.
The new \texttt{bz2} module is an interface to the bz2 data compression library. bz2-compressed data is usually smaller than corresponding \texttt{zlib}-compressed data. (Contributed by Gustavo Niemeyer.)

A set of standard date/time types has been added in the new \texttt{datetime} module. See the following section for more details.

The Distutils \texttt{Extension} class now supports an extra constructor argument named \texttt{depends} for listing additional source files that an extension depends on. This lets Distutils recompile the module if any of the dependency files are modified. For example, if \texttt{sampmodule.c} includes the header file \texttt{sample.h}, you would create the \texttt{Extension} object like this:

```python
ext = Extension("samp",
    sources=["sampmodule.c"],
    depends=["sample.h"])
```

Modifying \texttt{sample.h} would then cause the module to be recompiled. (Contributed by Jeremy Hylton.)

Other minor changes to Distutils: it now checks for the \texttt{cc}, \texttt{CFLAGS}, \texttt{CPP}, \texttt{LDFLAGS}, and \texttt{CPPFLAGS} environment variables, using them to override the settings in Python's configuration (contributed by Robert Weber).

Previously the \texttt{doctest} module would only search the docstrings of public methods and functions for test cases, but it now also examines private ones as well. The \texttt{DocTestSuite()} function creates a \texttt{unittest.TestSuite} object from a set of \texttt{doctest} tests.

The new \texttt{gc.get_referents(object)()} function returns a list of all the objects referenced by \texttt{object}. 
• The `getopt` module gained a new function, `gnu_getopt()`, that supports the same arguments as the existing `getopt()` function but uses GNU-style scanning mode. The existing `getopt()` stops processing options as soon as a non-option argument is encountered, but in GNU-style mode processing continues, meaning that options and arguments can be mixed. For example:

```python
>>> getopt.getopt(['-f', 'filename', 'output', '-v'], 'f:v')
([['-f', 'filename']], ['output', '-v'])

>>> getopt.gnu_getopt(['-f', 'filename', 'output', '-v'], 'f:v')
([['-f', 'filename'], ['-v', '']], ['output'])
```

(Contributed by Peter Åstrand.)

• The `grp`, `pwd`, and `resource` modules now return enhanced tuples:

```python
>>> import grp
>>> g = grp.getgrnam('amk')
>>> g.gr_name, g.gr_gid ('amk', 500)
```

• The `gzip` module can now handle files exceeding 2 GiB.

• The new `heapq` module contains an implementation of a heap queue algorithm. A heap is an array-like data structure that keeps items in a partially sorted order such that, for every index $k$, $\text{heap}[k] \leq \text{heap}[2k+1]$ and $\text{heap}[k] \leq \text{heap}[2k+2]$. This makes it quick to remove the smallest item, and inserting a new item while maintaining the heap property is $O(\log n)$. (See [https://xlinux.nist.gov/dads//HTML/priorityque.html](https://xlinux.nist.gov/dads//HTML/priorityque.html) for more information about the priority queue data structure.)

The `heapq` module provides `heappush()` and `heappop()` functions for adding and removing items while maintaining the heap
property on top of some other mutable Python sequence type. Here's an example that uses a Python list:

```python
>>> import heapq
>>> heap = []
>>> for item in [3, 7, 5, 11, 1]:
...    heapq.heappush(heap, item)
...
>>> heap
[1, 3, 5, 11, 7]
>>> heapq.heappop(heap)
1
>>> heapq.heappop(heap)
3
>>> heap
[5, 7, 11]
```

(Contributed by Kevin O’Connor.)

- The IDLE integrated development environment has been updated using the code from the IDLEfork project (http://idlefork.sourceforge.net). The most notable feature is that the code being developed is now executed in a subprocess, meaning that there's no longer any need for manual `reload()` operations. IDLE’s core code has been incorporated into the standard library as the `idlelib` package.

- The `imaplib` module now supports IMAP over SSL. (Contributed by Piers Lauder and Tino Lange.)

- The `itertools` contains a number of useful functions for use with iterators, inspired by various functions provided by the ML and Haskell languages. For example, `itertools.ifilter(predicate, iterator)` returns all elements in the iterator for which the function `predicate()` returns `True`, and `itertools.repeat(obj, N)` returns `obj` `N` times. There are a number of other functions in the module; see the package’s reference documentation for details. (Contributed by Raymond
Hettinger.)

- Two new functions in the `math` module, `degrees(rads)()` and `radians(degs)()`, convert between radians and degrees. Other functions in the `math` module such as `math.sin()` and `math.cos()` have always required input values measured in radians. Also, an optional `base` argument was added to `math.log()` to make it easier to compute logarithms for bases other than $e$ and $10$. (Contributed by Raymond Hettinger.)

- Several new POSIX functions (`getpgid()`, `killpg()`, `lchown()`, `loadavg()`, `major()`, `makedev()`, `minor()`, and `mknod()`) were added to the `posix` module that underlies the `os` module. (Contributed by Gustavo Niemeyer, Geert Jansen, and Denis S. Otkidach.)

- In the `os` module, the `*stat()` family of functions can now report fractions of a second in a timestamp. Such time stamps are represented as floats, similar to the value returned by `time.time()`.

  During testing, it was found that some applications will break if time stamps are floats. For compatibility, when using the tuple interface of the `stat_result` time stamps will be represented as integers. When using named fields (a feature first introduced in Python 2.2), time stamps are still represented as integers, unless `os.stat_float_times()` is invoked to enable float return values:

```
>>> os.stat("/tmp").st_mtime
1034791200
>>> os.stat_float_times(True)
>>> os.stat("/tmp").st_mtime
1034791200.6335014
```
In Python 2.4, the default will change to always returning floats.

Application developers should enable this feature only if all their libraries work properly when confronted with floating point time stamps, or if they use the tuple API. If used, the feature should be activated on an application level instead of trying to enable it on a per-use basis.

- The `optparse` module contains a new parser for command-line arguments that can convert option values to a particular Python type and will automatically generate a usage message. See the following section for more details.

- The old and never-documented `linuxaudiodev` module has been deprecated, and a new version named `ossaudiodev` has been added. The module was renamed because the OSS sound drivers can be used on platforms other than Linux, and the interface has also been tidied and brought up to date in various ways. (Contributed by Greg Ward and Nicholas FitzRoy-Dale.)

- The new `platform` module contains a number of functions that try to determine various properties of the platform you're running on. There are functions for getting the architecture, CPU type, the Windows OS version, and even the Linux distribution version. (Contributed by Marc-André Lemburg.)

- The parser objects provided by the `pyexpat` module can now optionally buffer character data, resulting in fewer calls to your character data handler and therefore faster performance. Setting the parser object's `buffer_text` attribute to `True` will enable buffering.

- The `sample(population, k)()` function was added to the `random` module. `population` is a sequence or `xrange` object containing
the elements of a population, and \texttt{sample()} chooses $k$ elements from the population without replacing chosen elements. $k$ can be any value up to \texttt{len(population)}. For example:

```python
>>> days = ['Mo', 'Tu', 'We', 'Th', 'Fr', 'St', 'Sn']
>>> random.sample(days, 3)  # Choose 3 elements
['St', 'Sn', 'Th']
>>> random.sample(days, 7)  # Choose 7 elements
['Tu', 'Th', 'Mo', 'We', 'St', 'Fr', 'Sn']
>>> random.sample(days, 7)  # Choose 7 again
['We', 'Mo', 'Sn', 'Fr', 'Tu', 'St', 'Th']
>>> random.sample(days, 8)  # Can't choose eight
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "random.py", line 414, in sample
    raise ValueError, "sample larger than population"
ValueError: sample larger than population
>>> random.sample(xrange(1,10000,2), 10)  # Choose ten odd
[3407, 3805, 1505, 7023, 2401, 2267, 9733, 3151, 8083, 9195]
```

The \texttt{random} module now uses a new algorithm, the Mersenne Twister, implemented in C. It's faster and more extensively studied than the previous algorithm.

(All changes contributed by Raymond Hettinger.)

- The \texttt{readline} module also gained a number of new functions: \texttt{get_history_item()}, \texttt{get_current_history_length()}, and \texttt{redisplay()}.

- The \texttt{rexec} and \texttt{Bastion} modules have been declared dead, and attempts to import them will fail with a \texttt{RuntimeError}. New-style classes provide new ways to break out of the restricted execution environment provided by \texttt{rexec}, and no one has interest in fixing them or time to do so. If you have applications using \texttt{rexec}, rewrite them to use something else.
(Sticking with Python 2.2 or 2.1 will not make your applications any safer because there are known bugs in the `rexec` module in those versions. To repeat: if you’re using `rexec`, stop using it immediately.)

- The `rotor` module has been deprecated because the algorithm it uses for encryption is not believed to be secure. If you need encryption, use one of the several AES Python modules that are available separately.

- The `shutil` module gained a `move(src, dest)()` function that recursively moves a file or directory to a new location.

- Support for more advanced POSIX signal handling was added to the `signal` but then removed again as it proved impossible to make it work reliably across platforms.

- The `socket` module now supports timeouts. You can call the `settimeout(t)()` method on a socket object to set a timeout of \( t \) seconds. Subsequent socket operations that take longer than \( t \) seconds to complete will abort and raise a `socket.timeout` exception.

  The original timeout implementation was by Tim O'Malley. Michael Gilfix integrated it into the Python `socket` module and shepherded it through a lengthy review. After the code was checked in, Guido van Rossum rewrote parts of it. (This is a good example of a collaborative development process in action.)

- On Windows, the `socket` module now ships with Secure Sockets Layer (SSL) support.

- The value of the C `PYTHON_API_VERSION` macro is now exposed at the Python level as `sys.api_version`. The current exception
can be cleared by calling the new `sys.exc_clear()` function.

- The new `tarfile` module allows reading from and writing to tar-format archive files. (Contributed by Lars Gustäbel.)

- The new `textwrap` module contains functions for wrapping strings containing paragraphs of text. The `wrap(text, width)()` function takes a string and returns a list containing the text split into lines of no more than the chosen width. The `fill(text, width)()` function returns a single string, reformatted to fit into lines no longer than the chosen width. (As you can guess, `fill()` is built on top of `wrap()`.) For example:

```python
>>> import textwrap
>>> paragraph = "Not a whit, we defy augury: ... more text
..."
>>> textwrap.wrap(paragraph, 60)
['"Not a whit, we defy augury: there's a special providence "
'the fall of a sparrow. If it be now, 'tis not to come; if
...']
>>> print textwrap.fill(paragraph, 35)
Not a whit, we defy augury: there's a special providence in the fall of
a sparrow. If it be now, 'tis not to come; if it be not to come, it
will be now; if it be not now, yet it will come: the readiness is all.
```  

The module also contains a `TextWrapper` class that actually implements the text wrapping strategy. Both the `TextWrapper` class and the `wrap()` and `fill()` functions support a number of additional keyword arguments for fine-tuning the formatting; consult the module’s documentation for details. (Contributed by Greg Ward.)

- The `thread` and `threading` modules now have companion modules, `dummy_thread` and `dummy_threading`, that provide a do-
nothing implementation of the thread module's interface for platforms where threads are not supported. The intention is to simplify thread-aware modules (ones that don't rely on threads to run) by putting the following code at the top:

```python
try:
    import threading as _threading
except ImportError:
    import dummy_threading as _threading
```

In this example, _threading is used as the module name to make it clear that the module being used is not necessarily the actual threading module. Code can call functions and use classes in _threading whether or not threads are supported, avoiding an if statement and making the code slightly clearer. This module will not magically make multithreaded code run without threads; code that waits for another thread to return or to do something will simply hang forever.

- The time module's strptime() function has long been an annoyance because it uses the platform C library's strptime() implementation, and different platforms sometimes have odd bugs. Brett Cannon contributed a portable implementation that's written in pure Python and should behave identically on all platforms.

- The new timeit module helps measure how long snippets of Python code take to execute. The timeit.py file can be run directly from the command line, or the module's Timer class can be imported and used directly. Here's a short example that figures out whether it's faster to convert an 8-bit string to Unicode by appending an empty Unicode string to it or by using the unicode() function:

```python
import timeit
```
The **Tix** module has received various bug fixes and updates for the current version of the Tix package.

The **Tkinter** module now works with a thread-enabled version of Tcl. Tcl's threading model requires that widgets only be accessed from the thread in which they're created; accesses from another thread can cause Tcl to panic. For certain Tcl interfaces, **Tkinter** will now automatically avoid this when a widget is accessed from a different thread by marshalling a command, passing it to the correct thread, and waiting for the results. Other interfaces can't be handled automatically but **Tkinter** will now raise an exception on such an access so that you can at least find out about the problem. See https://mail.python.org/pipermail/python-dev/2002-December/031107.html for a more detailed explanation of this change. (Implemented by Martin von Löwis.)

Calling Tcl methods through `_tkinter` no longer returns only strings. Instead, if Tcl returns other objects those objects are converted to their Python equivalent, if one exists, or wrapped with a `_tkinter.Tcl_Obj` object if no Python equivalent exists. This behavior can be controlled through the `wantobjects()` method of `tkapp` objects.
When using _tkinter through the Tkinter module (as most Tkinter applications will), this feature is always activated. It should not cause compatibility problems, since Tkinter would always convert string results to Python types where possible.

If any incompatibilities are found, the old behavior can be restored by setting the wantobjects variable in the Tkinter module to false before creating the first tkapp object.

```python
import Tkinter
Tkinter.wantobjects = 0
```

Any breakage caused by this change should be reported as a bug.

- The UserDict module has a new DictMixin class which defines all dictionary methods for classes that already have a minimum mapping interface. This greatly simplifies writing classes that need to be substitutable for dictionaries, such as the classes in the shelve module.

Adding the mix-in as a superclass provides the full dictionary interface whenever the class defines __getitem__(), __setitem__(), __delitem__(), and keys(). For example:

```python
>>> import UserDict
>>> class SeqDict(UserDict.DictMixin):
...     """Dictionary lookalike implemented with lists."""
...     def __init__(self):
...         self.keylist = []
...         self.valuelist = []
...     def __getitem__(self, key):
...         try:
...             i = self.keylist.index(key)
...         except ValueError:
...             raise KeyError
...         return self.valuelist[i]
...     def __setitem__(self, key, value):
```
try:
    i = self.keylist.index(key)
    self.valuelist[i] = value
except ValueError:
    self.keylist.append(key)
    self.valuelist.append(value)
def __delitem__(self, key):
    try:
        i = self.keylist.index(key)
    except ValueError:
        raise KeyError
    self.keylist.pop(i)
    self.valuelist.pop(i)
def keys(self):
    return list(self.keylist)

>>> s = SeqDict()
>>> dir(s) # See that other dictionary methods are implemented
['__cmp__', '__contains__', '__delitem__', '__doc__', '__ge__', '__init__', '__iter__', '__len__', '__module__', '__repr__', '__setitem__', 'clear', 'get', 'has_key', 'items', 'iteritems', 'iterkeys', 'itervalues', 'keylist', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'valuelist', 'values']

(Contributed by Raymond Hettinger.)

- The DOM implementation in `xml.dom.minidom` can now generate XML output in a particular encoding by providing an optional encoding argument to the `toxml()` and `toprettyxml()` methods of DOM nodes.

- The `xmlrpclib` module now supports an XML-RPC extension for handling nil data values such as Python's `None`. Nil values are always supported on unmarshalling an XML-RPC response. To generate requests containing `None`, you must supply a true value for the `allow_none` parameter when creating a `Marshaller` instance.

- The new `DocXMLRPCServer` module allows writing self-documenting XML-RPC servers. Run it in demo mode (as a
program) to see it in action. Pointing the Web browser to the RPC server produces pydoc-style documentation; pointing xmlrpclib to the server allows invoking the actual methods. (Contributed by Brian Quinlan.)

- Support for internationalized domain names (RFCs 3454, 3490, 3491, and 3492) has been added. The “idna” encoding can be used to convert between a Unicode domain name and the ASCII-compatible encoding (ACE) of that name.

```
>{}<> u"www.Alliancefrançaise.nu".encode("idna")
'www.xn--alliancefranaise-npb.nu'
```

The `socket` module has also been extended to transparently convert Unicode hostnames to the ACE version before passing them to the C library. Modules that deal with hostnames such as `httplib` and `ftplib`) also support Unicode host names; `httplib` also sends HTTP `Host` headers using the ACE version of the domain name. `urllib` supports Unicode URLs with non-ASCII host names as long as the `path` part of the URL is ASCII only.

To implement this change, the `stringprep` module, the `mkstringprep` tool and the `punycode` encoding have been added.

**Date/Time Type**

Date and time types suitable for expressing timestamps were added as the `datetime` module. The types don't support different calendars or many fancy features, and just stick to the basics of representing time.

The three primary types are: `date`, representing a day, month, and year; `time`, consisting of hour, minute, and second; and `datetime`, which contains all the attributes of both `date` and `time`. There's also
a **timedelta** class representing differences between two points in time, and time zone logic is implemented by classes inheriting from the abstract **tzinfo** class.

You can create instances of **date** and **time** by either supplying keyword arguments to the appropriate constructor, e.g. `datetime.date(year=1972, month=10, day=15)`, or by using one of a number of class methods. For example, the `date.today()` class method returns the current local date.

Once created, instances of the date/time classes are all immutable. There are a number of methods for producing formatted strings from objects:

```python
>>> import datetime
>>> now = datetime.datetime.now()
>>> now.isoformat()
'2002-12-30T21:27:03.994956'
>>> now.ctime()  # Only available on date, datetime
'Mon Dec 30 21:27:03 2002'
>>> now.strftime('%Y %d %b')
'2002 30 Dec'
```

The `replace()` method allows modifying one or more fields of a **date** or **datetime** instance, returning a new instance:

```python
>>> d = datetime.datetime.now()
>>> d
datetime.datetime(2002, 12, 30, 22, 15, 38, 827738)
>>> d.replace(year=2001, hour = 12)
datetime.datetime(2001, 12, 30, 12, 15, 38, 827738)
```

Instances can be compared, hashed, and converted to strings (the result is the same as that of `isoformat()`). **date** and **datetime** instances can be subtracted from each other, and added to **timedelta** instances. The largest missing feature is that there's no
standard library support for parsing strings and getting back a date or datetime.

For more information, refer to the module's reference documentation. (Contributed by Tim Peters.)

The optparse Module

The getopt module provides simple parsing of command-line arguments. The new optparse module (originally named Optik) provides more elaborate command-line parsing that follows the Unix conventions, automatically creates the output for --help, and can perform different actions for different options.

You start by creating an instance of OptionParser and telling it what your program’s options are.

```python
import sys
from optparse import OptionParser

op = OptionParser()
op.add_option('-i', '--input',
             action='store', type='string', dest='input',
             help='set input filename')

op.add_option('-l', '--length',
             action='store', type='int', dest='length',
             help='set maximum length of output')

options, args = op.parse_args(sys.argv[1:])
print options
print args
```

Parsing a command line is then done by calling the parse_args() method.

This returns an object containing all of the option values, and a list of strings containing the remaining arguments.
Invoking the script with the various arguments now works as you’d expect it to. Note that the length argument is automatically converted to an integer.

```bash
$ ./python opt.py -i data arg1
<Values at 0x400cad4c: {'input': 'data', 'length': None}>
['arg1']
$ ./python opt.py --input=data --length=4
<Values at 0x400cad2c: {'input': 'data', 'length': 4}>
[]
```

The help message is automatically generated for you:

```bash
$ ./python opt.py --help
usage: opt.py [options]

options:
  -h, --help          show this help message and exit
  -iINPUT, --input=INPUT
                      set input filename
  -lLENGTH, --length=LENGTH
                      set maximum length of output
```

See the module’s documentation for more details.

Optik was written by Greg Ward, with suggestions from the readers of the Getopt SIG.
Pymalloc: A Specialized Object Allocator

Pymalloc, a specialized object allocator written by Vladimir Marangozov, was a feature added to Python 2.1. Pymalloc is intended to be faster than the system `malloc()` and to have less memory overhead for allocation patterns typical of Python programs. The allocator uses C’s `malloc()` function to get large pools of memory and then fulfills smaller memory requests from these pools.

In 2.1 and 2.2, pymalloc was an experimental feature and wasn’t enabled by default; you had to explicitly enable it when compiling Python by providing the `--with-pymalloc` option to the `configure` script. In 2.3, pymalloc has had further enhancements and is now enabled by default; you’ll have to supply `--without-pymalloc` to disable it.

This change is transparent to code written in Python; however, pymalloc may expose bugs in C extensions. Authors of C extension modules should test their code with pymalloc enabled, because some incorrect code may cause core dumps at runtime.

There’s one particularly common error that causes problems. There are a number of memory allocation functions in Python's C API that have previously just been aliases for the C library’s `malloc()` and `free()`, meaning that if you accidentally called mismatched functions the error wouldn’t be noticeable. When the object allocator is enabled, these functions aren’t aliases of `malloc()` and `free()` any more, and calling the wrong function to free memory may get you a core dump. For example, if memory was allocated using `PyObject_Malloc()`, it has to be freed using `PyObject_Free()`, not `free()`. A few modules included with Python fell afoul of this and had to be fixed; doubtless there are more third-party modules that will...
have the same problem.

As part of this change, the confusing multiple interfaces for allocating memory have been consolidated down into two API families. Memory allocated with one family must not be manipulated with functions from the other family. There is one family for allocating chunks of memory and another family of functions specifically for allocating Python objects.

- To allocate and free an undistinguished chunk of memory use the “raw memory” family: `PyMem_Malloc()`, `PyMem_Realloc()`, and `PyMem_Free()`.
- The “object memory” family is the interface to the pymalloc facility described above and is biased towards a large number of “small” allocations: `PyObject_Malloc()`, `PyObject_Realloc()`, and `PyObject_Free()`.
- To allocate and free Python objects, use the “object” family `PyObject_New()`, `PyObject_NewVar()`, and `PyObject_Del()`.

Thanks to lots of work by Tim Peters, pymalloc in 2.3 also provides debugging features to catch memory overwrites and doubled frees in both extension modules and in the interpreter itself. To enable this support, compile a debugging version of the Python interpreter by running `configure` with `--with-pydebug`.

To aid extension writers, a header file `Misc/pymemcompat.h` is distributed with the source to Python 2.3 that allows Python extensions to use the 2.3 interfaces to memory allocation while compiling against any version of Python since 1.5.2. You would copy the file from Python’s source distribution and bundle it with the source of your extension.

See also:

https://hg.python.org/cpython/file/default/Objects/obmalloc.c
For the full details of the pymalloc implementation, see the comments at the top of the file `Objects/obmalloc.c` in the Python source code. The above link points to the file within the python.org SVN browser.
Build and C API Changes

Changes to Python's build process and to the C API include:

- The cycle detection implementation used by the garbage collection has proven to be stable, so it's now been made mandatory. You can no longer compile Python without it, and the --with-cycle-gc switch to configure has been removed.

- Python can now optionally be built as a shared library (libpython2.3.so) by supplying --enable-shared when running Python's configure script. (Contributed by Ondrej Palkovsky.)

- The DL_EXPORT and DL_IMPORT macros are now deprecated. Initialization functions for Python extension modules should now be declared using the new macro PyMODINIT_FUNC, while the Python core will generally use the PyAPI_FUNC and PyAPI_DATA macros.

- The interpreter can be compiled without any docstrings for the built-in functions and modules by supplying --without-docstrings to the configure script. This makes the Python executable about 10% smaller, but will also mean that you can't get help for Python's built-ins. (Contributed by Gustavo Niemeyer.)

- The PyArg_NoArgs() macro is now deprecated, and code that uses it should be changed. For Python 2.2 and later, the method definition table can specify the METH_NOARGS flag, signalling that there are no arguments, and the argument checking can then be removed. If compatibility with pre-2.2 versions of Python is important, the code could use PyArg_ParseTuple(args, "") instead, but this will be slower than using METH_NOARGS.

- PyArg_ParseTuple() accepts new format characters for various sizes of unsigned integers: B for unsigned char, H for unsigned...
short int, I for unsigned int, and k for unsigned long long.

- A new function, `PyObject_DelItemString(mapping, char *key)()` was added as shorthand for `PyObject_DelItem(mapping, PyString_New(key))`.

- File objects now manage their internal string buffer differently, increasing it exponentially when needed. This results in the benchmark tests in `Lib/test/test_bufio.py` speeding up considerably (from 57 seconds to 1.7 seconds, according to one measurement).

- It's now possible to define class and static methods for a C extension type by setting either the `METH_CLASS` or `METH_STATIC` flags in a method's `PyMethodDef` structure.

- Python now includes a copy of the Expat XML parser's source code, removing any dependence on a system version or local installation of Expat.

- If you dynamically allocate type objects in your extension, you should be aware of a change in the rules relating to the `__module__` and `__name__` attributes. In summary, you will want to ensure the type's dictionary contains a '__module__' key; making the module name the part of the type name leading up to the final period will no longer have the desired effect. For more detail, read the API reference documentation or the source.

### Port-Specific Changes

Support for a port to IBM’s OS/2 using the EMX runtime environment was merged into the main Python source tree. EMX is a POSIX emulation layer over the OS/2 system APIs. The Python port for EMX tries to support all the POSIX-like capability exposed by the EMX runtime, and mostly succeeds; `fork()` and `fcntl()` are restricted by the limitations of the underlying emulation layer. The standard OS/2 port, which uses IBM’s Visual Age compiler, also
gained support for case-sensitive import semantics as part of the integration of the EMX port into CVS. (Contributed by Andrew MacIntyre.)

On MacOS, most toolbox modules have been weaklinked to improve backward compatibility. This means that modules will no longer fail to load if a single routine is missing on the current OS version. Instead calling the missing routine will raise an exception. (Contributed by Jack Jansen.)

The RPM spec files, found in the Misc/RPM/ directory in the Python source distribution, were updated for 2.3. (Contributed by Sean Reifschneider.)

Other new platforms now supported by Python include AtheOS (http://atheos.cx/), GNU/Hurd, and OpenVMS.
Other Changes and Fixes

As usual, there were a bunch of other improvements and bugfixes scattered throughout the source tree. A search through the CVS change logs finds there were 523 patches applied and 514 bugs fixed between Python 2.2 and 2.3. Both figures are likely to be underestimates.

Some of the more notable changes are:

- If the \texttt{PYTHONINSPECT} environment variable is set, the Python interpreter will enter the interactive prompt after running a Python program, as if Python had been invoked with the \texttt{-i} option. The environment variable can be set before running the Python interpreter, or it can be set by the Python program as part of its execution.

- The \texttt{regrtest.py} script now provides a way to allow “all resources except foo.” A resource name passed to the \texttt{-u} option can now be prefixed with a hyphen ('-') to mean “remove this resource.” For example, the option \texttt{"-uall,-bsddb"} could be used to enable the use of all resources except \texttt{bsddb}.

- The tools used to build the documentation now work under Cygwin as well as Unix.

- The \texttt{SET_LINENO} opcode has been removed. Back in the mists of time, this opcode was needed to produce line numbers in tracebacks and support trace functions (for, e.g., \texttt{pdb}). Since Python 1.5, the line numbers in tracebacks have been computed using a different mechanism that works with “python -O”. For Python 2.3 Michael Hudson implemented a similar scheme to determine when to call the trace function, removing the need for
It would be difficult to detect any resulting difference from Python code, apart from a slight speed up when Python is run without `-O`.

C extensions that access the `f_lineno` field of frame objects should instead call `PyCode_Addr2Line(f->f_code, f->f_lasti)`. This will have the added effect of making the code work as desired under “python -O” in earlier versions of Python.

A nifty new feature is that trace functions can now assign to the `f_lineno` attribute of frame objects, changing the line that will be executed next. A `jump` command has been added to the `pdb` debugger taking advantage of this new feature. (Implemented by Richie Hindle.)
Porting to Python 2.3

This section lists previously described changes that may require changes to your code:

- **yield** is now always a keyword; if it's used as a variable name in your code, a different name must be chosen.

- For strings $X$ and $Y$, $X$ in $Y$ now works if $X$ is more than one character long.

- The **int()** type constructor will now return a long integer instead of raising an **OverflowError** when a string or floating-point number is too large to fit into an integer.

- If you have Unicode strings that contain 8-bit characters, you must declare the file's encoding (UTF-8, Latin-1, or whatever) by adding a comment to the top of the file. See section PEP 263: **Source Code Encodings** for more information.

- Calling Tcl methods through **_tkinter** no longer returns only strings. Instead, if Tcl returns other objects those objects are converted to their Python equivalent, if one exists, or wrapped with a **_tkinter.Tcl_Obj** object if no Python equivalent exists.

- Large octal and hex literals such as **0xffffffff** now trigger a **FutureWarning**. Currently they're stored as 32-bit numbers and result in a negative value, but in Python 2.4 they'll become positive long integers.

There are a few ways to fix this warning. If you really need a positive number, just add an **L** to the end of the literal. If you're trying to get a 32-bit integer with low bits set and have
previously used an expression such as \(-(1 \ll 31)\), it's probably clearest to start with all bits set and clear the desired upper bits. For example, to clear just the top bit (bit 31), you could write 0xffffffffl \&(1l\ll31).

- You can no longer disable assertions by assigning to \_debug\_.

- The Distutils `setup()` function has gained various new keyword arguments such as `depends`. Old versions of the Distutils will abort if passed unknown keywords. A solution is to check for the presence of the new `get_distutil_options()` function in your `setup.py` and only uses the new keywords with a version of the Distutils that supports them:

```python
from distutils import core
kw = {'sources': 'foo.c', ...
if hasattr(core, 'get_distutil_options'):
    kw['depends'] = ['foo.h']
ext = Extension(**kw)
```

- Using `None` as a variable name will now result in a `SyntaxWarning` warning.

- Names of extension types defined by the modules included with Python now contain the module and a \'.\' in front of the type name.
Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Jeff Bauer, Simon Brunning, Brett Cannon, Michael Chermside, Andrew Dalke, Scott David Daniels, Fred L. Drake, Jr., David Fraser, Kelly Gerber, Raymond Hettinger, Michael Hudson, Chris Lambert, Detlef Lannert, Martin von Löwis, Andrew MacIntyre, Lalo Martins, Chad Netzer, Gustavo Niemeyer, Neal Norwitz, Hans Nowak, Chris Reedy, Francesco Ricciardi, Vinay Sajip, Neil Schemenauer, Roman Suzi, Jason Tishler, Just van Rossum.
What’s New in Python 2.2

Author: A.M. Kuchling
Introduction

This article explains the new features in Python 2.2.2, released on October 14, 2002. Python 2.2.2 is a bugfix release of Python 2.2, originally released on December 21, 2001.

Python 2.2 can be thought of as the “cleanup release”. There are some features such as generators and iterators that are completely new, but most of the changes, significant and far-reaching though they may be, are aimed at cleaning up irregularities and dark corners of the language design.

This article doesn’t attempt to provide a complete specification of the new features, but instead provides a convenient overview. For full details, you should refer to the documentation for Python 2.2, such as the Python Library Reference and the Python Reference Manual. If you want to understand the complete implementation and design rationale for a change, refer to the PEP for a particular new feature.
The largest and most far-reaching changes in Python 2.2 are to Python's model of objects and classes. The changes should be backward compatible, so it's likely that your code will continue to run unchanged, but the changes provide some amazing new capabilities. Before beginning this, the longest and most complicated section of this article, I'll provide an overview of the changes and offer some comments.

A long time ago I wrote a Web page listing flaws in Python's design. One of the most significant flaws was that it's impossible to subclass Python types implemented in C. In particular, it's not possible to subclass built-in types, so you can't just subclass, say, lists in order to add a single useful method to them. The UserList module provides a class that supports all of the methods of lists and that can be subclassed further, but there's lots of C code that expects a regular Python list and won't accept a UserList instance.

Python 2.2 fixes this, and in the process adds some exciting new capabilities. A brief summary:

- You can subclass built-in types such as lists and even integers, and your subclasses should work in every place that requires the original type.
- It's now possible to define static and class methods, in addition to the instance methods available in previous versions of Python.
- It's also possible to automatically call methods on accessing or setting an instance attribute by using a new mechanism called properties. Many uses of __getattr__() can be rewritten to use properties instead, making the resulting code simpler and faster.
As a small side benefit, attributes can now have docstrings, too.

- The list of legal attributes for an instance can be limited to a particular set using *slots*, making it possible to safeguard against typos and perhaps make more optimizations possible in future versions of Python.

Some users have voiced concern about all these changes. Sure, they say, the new features are neat and lend themselves to all sorts of tricks that weren't possible in previous versions of Python, but they also make the language more complicated. Some people have said that they've always recommended Python for its simplicity, and feel that its simplicity is being lost.

Personally, I think there's no need to worry. Many of the new features are quite esoteric, and you can write a lot of Python code without ever needed to be aware of them. Writing a simple class is no more difficult than it ever was, so you don't need to bother learning or teaching them unless they're actually needed. Some very complicated tasks that were previously only possible from C will now be possible in pure Python, and to my mind that's all for the better.

I'm not going to attempt to cover every single corner case and small change that were required to make the new features work. Instead this section will paint only the broad strokes. See section Related Links, “Related Links”, for further sources of information about Python 2.2’s new object model.

**Old and New Classes**

First, you should know that Python 2.2 really has two kinds of classes: classic or old-style classes, and new-style classes. The old-style class model is exactly the same as the class model in earlier versions of Python. All the new features described in this section apply only to new-style classes. This divergence isn't intended to last forever; eventually old-style classes will be dropped, possibly in
Python 3.0.

So how do you define a new-style class? You do it by subclassing an existing new-style class. Most of Python’s built-in types, such as integers, lists, dictionaries, and even files, are new-style classes now. A new-style class named `object`, the base class for all built-in types, has also been added so if no built-in type is suitable, you can just subclass `object`:

```python
class C(object):
    def __init__(self):
        ...
```

This means that `class` statements that don’t have any base classes are always classic classes in Python 2.2. (Actually you can also change this by setting a module-level variable named `__metaclass__` — see PEP 253 for the details — but it’s easier to just subclass `object`.)

The type objects for the built-in types are available as built-ins, named using a clever trick. Python has always had built-in functions named `int()`, `float()`, and `str()`. In 2.2, they aren’t functions any more, but type objects that behave as factories when called:

```python
>>> int
<type 'int'>
>>> int('123')
123
```

To make the set of types complete, new type objects such as `dict()` and `file()` have been added. Here’s a more interesting example, adding a `lock()` method to file objects:

```python
class LockableFile(file):
    def lock(self, operation, length=0, start=0, whence=0):
        import fcntl
```
return fcntl.lockf(self.fileno(), operation, length, start, whence)

The now-obsolete `posixfile` module contained a class that emulated all of a file object's methods and also added a `lock()` method, but this class couldn't be passed to internal functions that expected a built-in file, something which is possible with our new `LockableFile`.

## Descriptors

In previous versions of Python, there was no consistent way to discover what attributes and methods were supported by an object. There were some informal conventions, such as defining `__members__` and `__methods__` attributes that were lists of names, but often the author of an extension type or a class wouldn't bother to define them. You could fall back on inspecting the `__dict__` of an object, but when class inheritance or an arbitrary `__getattr__()` hook were in use this could still be inaccurate.

The one big idea underlying the new class model is that an API for describing the attributes of an object using *descriptors* has been formalized. Descriptors specify the value of an attribute, stating whether it's a method or a field. With the descriptor API, static methods and class methods become possible, as well as more exotic constructs.

Attribute descriptors are objects that live inside class objects, and have a few attributes of their own:

- `__name__` is the attribute's name.
- `__doc__` is the attribute's docstring.
- `__get__(object)()` is a method that retrieves the attribute value from `object`. 
• __set__(object, value)() sets the attribute on object to value.
• __delete__(object, value)() deletes the value attribute of object.

For example, when you write obj.x, the steps that Python actually performs are:

```python
descriptor = obj.__class__.x
descriptor.__get__(obj)
```

For methods, descriptor.__get__() returns a temporary object that's callable, and wraps up the instance and the method to be called on it. This is also why static methods and class methods are now possible; they have descriptors that wrap up just the method, or the method and the class. As a brief explanation of these new kinds of methods, static methods aren't passed the instance, and therefore resemble regular functions. Class methods are passed the class of the object, but not the object itself. Static and class methods are defined like this:

```python
class C(object):
    def f(arg1, arg2):
        ...
    f = staticmethod(f)

    def g(cls, arg1, arg2):
        ...
    g = classmethod(g)
```

The staticmethod() function takes the function f(), and returns it wrapped up in a descriptor so it can be stored in the class object. You might expect there to be special syntax for creating such methods (def static f, defstatic f(), or something like that) but no such syntax has been defined yet; that's been left for future versions of Python.

More new features, such as slots and properties, are also
implemented as new kinds of descriptors, and it's not difficult to write
a descriptor class that does something novel. For example, it would
be possible to write a descriptor class that made it possible to write
Eiffel-style preconditions and postconditions for a method. A class
that used this feature might be defined like this:

```python
from eiffel import eiffelmethod
class C(object):
    def f(self, arg1, arg2):
        # The actual function
        ...
    def pre_f(self):
        # Check preconditions
        ...
    def post_f(self):
        # Check postconditions
        ...
    f = eiffelmethod(f, pre_f, post_f)
```

Note that a person using the new `eiffelmethod()` doesn't have to
understand anything about descriptors. This is why I think the new
features don't increase the basic complexity of the language. There
will be a few wizards who need to know about it in order to write
`eiffelmethod()` or the ZODB or whatever, but most users will just
write code on top of the resulting libraries and ignore the
implementation details.

**Multiple Inheritance: The Diamond Rule**

Multiple inheritance has also been made more useful through
changing the rules under which names are resolved. Consider this
set of classes (diagram taken from PEP 253 by Guido van Rossum):

```
class A:
    ^   ^  def save(self): ...
    / \ |
    /  \
```
The lookup rule for classic classes is simple but not very smart; the base classes are searched depth-first, going from left to right. A reference to `D.save()` will search the classes `D`, `B`, and then `A`, where `save()` would be found and returned. `C.save()` would never be found at all. This is bad, because if `c`'s `save()` method is saving some internal state specific to `c`, not calling it will result in that state never getting saved.

New-style classes follow a different algorithm that's a bit more complicated to explain, but does the right thing in this situation. (Note that Python 2.3 changes this algorithm to one that produces the same results in most cases, but produces more useful results for really complicated inheritance graphs.)

1. List all the base classes, following the classic lookup rule and include a class multiple times if it's visited repeatedly. In the above example, the list of visited classes is `[D, B, A, C, A]`.
2. Scan the list for duplicated classes. If any are found, remove all but one occurrence, leaving the last one in the list. In the above example, the list becomes `[D, B, C, A]` after dropping duplicates.

Following this rule, referring to `D.save()` will return `C.save()`, which is the behaviour we're after. This lookup rule is the same as the one followed by Common Lisp. A new built-in function, `super()`, provides a way to get at a class's superclasses without having to reimplement Python's algorithm. The most commonly used form will be
super(class, obj)(), which returns a bound superclass object (not the actual class object). This form will be used in methods to call a method in the superclass; for example, D's save() method would look like this:

```python
class D (B, C):
    def save (self):
        # Call superclass .save()
        super(D, self).save()
        # Save D's private information here
        ...
```

super() can also return unbound superclass objects when called as super(class)() or super(class1, class2)(), but this probably won’t often be useful.

Attribute Access

A fair number of sophisticated Python classes define hooks for attribute access using __getattr__() ; most commonly this is done for convenience, to make code more readable by automatically mapping an attribute access such as obj.parent into a method call such as obj.get_parent . Python 2.2 adds some new ways of controlling attribute access.

First, __getattr__(attr_name)() is still supported by new-style classes, and nothing about it has changed. As before, it will be called when an attempt is made to access obj.foo and no attribute named foo is found in the instance’s dictionary.

New-style classes also support a new method, __getattribute__(attr_name)(). The difference between the two methods is that __getattribute__() is always called whenever any attribute is accessed, while the old __getattr__() is only called if foo isn’t found in the instance’s dictionary.
However, Python 2.2’s support for properties will often be a simpler way to trap attribute references. Writing a \texttt{__getattr__()} method is complicated because to avoid recursion you can’t use regular attribute accesses inside them, and instead have to mess around with the contents of \texttt{__dict__}. \texttt{__getattr__()} methods also end up being called by Python when it checks for other methods such as \texttt{__repr__()} or \texttt{__coerce__()}, and so have to be written with this in mind. Finally, calling a function on every attribute access results in a sizable performance loss.

\texttt{property} is a new built-in type that packages up three functions that get, set, or delete an attribute, and a docstring. For example, if you want to define a \texttt{size} attribute that’s computed, but also settable, you could write:

```python
class C(object):
    def get_size (self):
        result = ... computation ...
        return result
    def set_size (self, size):
        ... compute something based on the size
        and set internal state appropriately ...

    # Define a property. The 'delete this attribute'
    # method is defined as None, so the attribute
    # can't be deleted.
    size = property(get_size, set_size,
                    None,
                    "Storage size of this instance")
```

That is certainly clearer and easier to write than a pair of \texttt{__getattr__()}/\texttt{__setattr__()} methods that check for the \texttt{size} attribute and handle it specially while retrieving all other attributes from the instance’s \texttt{__dict__}. Accesses to \texttt{size} are also the only ones which have to perform the work of calling a function, so references to other attributes run at their usual speed.

Finally, it’s possible to constrain the list of attributes that can be
referenced on an object using the new __slots__ class attribute. Python objects are usually very dynamic; at any time it's possible to define a new attribute on an instance by just doing `obj.new_attr=1`. A new-style class can define a class attribute named __slots__ to limit the legal attributes to a particular set of names. An example will make this clear:

```python
>>> class C(object):
...     __slots__ = ('template', 'name')
...
>>> obj = C()
>>> print obj.template
None
>>> obj.template = 'Test'
>>> print obj.template
Test
>>> obj.newattr = None
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'C' object has no attribute 'newattr'
```

Note how you get an AttributeError on the attempt to assign to an attribute not listed in __slots__.

### Related Links

This section has just been a quick overview of the new features, giving enough of an explanation to start you programming, but many details have been simplified or ignored. Where should you go to get a more complete picture?

https://docs.python.org/dev/howto/descriptor.html is a lengthy tutorial introduction to the descriptor features, written by Guido van Rossum. If my description has whetted your appetite, go read this tutorial next, because it goes into much more detail about the new features while still remaining quite easy to read.
Next, there are two relevant PEPs, **PEP 252** and **PEP 253**. **PEP 252** is titled “Making Types Look More Like Classes”, and covers the descriptor API. **PEP 253** is titled “Subtyping Built-in Types”, and describes the changes to type objects that make it possible to subtype built-in objects. **PEP 253** is the more complicated PEP of the two, and at a few points the necessary explanations of types and meta-types may cause your head to explode. Both PEPs were written and implemented by Guido van Rossum, with substantial assistance from the rest of the Zope Corp. team.

Finally, there’s the ultimate authority: the source code. Most of the machinery for the type handling is in `Objects/typeobject.c`, but you should only resort to it after all other avenues have been exhausted, including posting a question to python-list or python-dev.
Another significant addition to 2.2 is an iteration interface at both the C and Python levels. Objects can define how they can be looped over by callers.

In Python versions up to 2.1, the usual way to make `for item in obj` work is to define a `__getitem__() ` method that looks something like this:

```python
def __getitem__(self, index):
    return <next item>
```

`__getitem__() ` is more properly used to define an indexing operation on an object so that you can write `obj[5]` to retrieve the sixth element. It’s a bit misleading when you’re using this only to support `for` loops. Consider some file-like object that wants to be looped over; the `index ` parameter is essentially meaningless, as the class probably assumes that a series of `__getitem__() ` calls will be made with `index ` incrementing by one each time. In other words, the presence of the `__getitem__() ` method doesn’t mean that using `file[5]` to randomly access the sixth element will work, though it really should.

In Python 2.2, iteration can be implemented separately, and `__getitem__() ` methods can be limited to classes that really do support random access. The basic idea of iterators is simple. A new built-in function, `iter(obj)()` or `iter(C, sentinel)`, is used to get an iterator. `iter(obj)()` returns an iterator for the object `obj`, while `iter(C, sentinel) ` returns an iterator that will invoke the callable object `C ` until it returns `sentinel ` to signal that the iterator is done.

Python classes can define an `__iter__() ` method, which should
create and return a new iterator for the object; if the object is its own iterator, this method can just return \texttt{self}. In particular, iterators will usually be their own iterators. Extension types implemented in C can implement a \texttt{tp_iter} function in order to return an iterator, and extension types that want to behave as iterators can define a \texttt{tp_iternext} function.

So, after all this, what do iterators actually do? They have one required method, \texttt{next()}, which takes no arguments and returns the next value. When there are no more values to be returned, calling \texttt{next()} should raise the \texttt{StopIteration} exception.

```python
>>> L = [1, 2, 3]
>>> i = iter(L)
>>> print i
<iterator object at 0x8116870>
>>> i.next()
1
>>> i.next()
2
>>> i.next()
3
>>> i.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
```

In 2.2, Python's \texttt{for} statement no longer expects a sequence; it expects something for which \texttt{iter()} will return an iterator. For backward compatibility and convenience, an iterator is automatically constructed for sequences that don't implement \texttt{__iter__()} or a \texttt{tp_iter} slot, so \texttt{for i in [1, 2, 3]} will still work. Wherever the Python interpreter loops over a sequence, it's been changed to use the iterator protocol. This means you can do things like this:

```python
>>> L = [1, 2, 3]
>>> i = iter(L)
```
Iterator support has been added to some of Python’s basic types. Calling `iter()` on a dictionary will return an iterator which loops over its keys:

```python
>>> m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'Jun ...
    'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11, 'D
>>> for key in m: print(key, m[key])
  ... Mar 3
  Feb 2
  Aug 8
  Sep 9
  May 5
  Jun 6
  Jul 7
  Jan 1
  Apr 4
  Nov 11
  Dec 12
  Oct 10
```

That’s just the default behaviour. If you want to iterate over keys, values, or key/value pairs, you can explicitly call the `iterkeys()`, `itervalues()`, or `iteritems()` methods to get an appropriate iterator. In a minor related change, the `in` operator now works on dictionaries, so `key in dict` is now equivalent to `dict.has_key(key)`.

Files also provide an iterator, which calls the `readline()` method until there are no more lines in the file. This means you can now read each line of a file using code like this:

```python
for line in file:
    # do something for each line
...
Note that you can only go forward in an iterator; there's no way to get the previous element, reset the iterator, or make a copy of it. An iterator object could provide such additional capabilities, but the iterator protocol only requires a `next()` method.

See also:

**PEP 234 - Iterators**

Written by Ka-Ping Yee and GvR; implemented by the Python Labs crew, mostly by GvR and Tim Peters.
Generators are another new feature, one that interacts with the introduction of iterators.

You’re doubtless familiar with how function calls work in Python or C. When you call a function, it gets a private namespace where its local variables are created. When the function reaches a `return` statement, the local variables are destroyed and the resulting value is returned to the caller. A later call to the same function will get a fresh new set of local variables. But, what if the local variables weren’t thrown away on exiting a function? What if you could later resume the function where it left off? This is what generators provide; they can be thought of as resumable functions.

Here’s the simplest example of a generator function:

```python
def generate_ints(N):
    for i in range(N):
        yield i
```

A new keyword, `yield`, was introduced for generators. Any function containing a `yield` statement is a generator function; this is detected by Python’s bytecode compiler which compiles the function specially as a result. Because a new keyword was introduced, generators must be explicitly enabled in a module by including a `from __future__ import generators` statement near the top of the module’s source code. In Python 2.3 this statement will become unnecessary.

When you call a generator function, it doesn’t return a single value; instead it returns a generator object that supports the iterator protocol. On executing the `yield` statement, the generator outputs the value of `i`, similar to a `return` statement. The big difference
between `yield` and a `return` statement is that on reaching a `yield` the generator’s state of execution is suspended and local variables are preserved. On the next call to the generator’s `next()` method, the function will resume executing immediately after the `yield` statement. (For complicated reasons, the `yield` statement isn't allowed inside the `try` block of a `try...finally` statement; read PEP 255 for a full explanation of the interaction between `yield` and exceptions.)

Here’s a sample usage of the `generate_ints()` generator:

```python
>>> gen = generate_ints(3)
>>> gen
<generator object at 0x8117f90>
>>> gen.next()
0
>>> gen.next()
1
>>> gen.next()
2
>>> gen.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in generate_ints
StopIteration
```

You could equally write `for i in generate_ints(5)`, or `a, b, c = generate_ints(3)`.

Inside a generator function, the `return` statement can only be used without a value, and signals the end of the procession of values; afterwards the generator cannot return any further values. `return` with a value, such as `return 5`, is a syntax error inside a generator function. The end of the generator’s results can also be indicated by raising `StopIteration` manually, or by just letting the flow of execution fall off the bottom of the function.
You could achieve the effect of generators manually by writing your own class and storing all the local variables of the generator as instance variables. For example, returning a list of integers could be done by setting `self.count` to 0, and having the `next()` method increment `self.count` and return it. However, for a moderately complicated generator, writing a corresponding class would be much messier. `Lib/test/test_generators.py` contains a number of more interesting examples. The simplest one implements an in-order traversal of a tree using generators recursively.

```python
# A recursive generator that generates Tree leaves in in-order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x
        yield t.label
        for x in inorder(t.right):
            yield x
```

Two other examples in `Lib/test/test_generators.py` produce solutions for the N-Queens problem (placing \(N\) queens on an \(N\times N\) chess board so that no queen threatens another) and the Knight’s Tour (a route that takes a knight to every square of an \(N\times N\) chessboard without visiting any square twice).

The idea of generators comes from other programming languages, especially Icon ([https://www.cs.arizona.edu/icon/](https://www.cs.arizona.edu/icon/)), where the idea of generators is central. In Icon, every expression and function call behaves like a generator. One example from “An Overview of the Icon Programming Language” at [https://www.cs.arizona.edu/icon/docs/ipd266.htm](https://www.cs.arizona.edu/icon/docs/ipd266.htm) gives an idea of what this looks like:

```icon
sentence := "Store it in the neighboring harbor"
if (i := find("or", sentence)) > 5 then write(i)
```
In Icon the `find()` function returns the indexes at which the substring “or” is found: 3, 23, 33. In the `if` statement, `i` is first assigned a value of 3, but 3 is less than 5, so the comparison fails, and Icon retries it with the second value of 23. 23 is greater than 5, so the comparison now succeeds, and the code prints the value 23 to the screen.

Python doesn’t go nearly as far as Icon in adopting generators as a central concept. Generators are considered a new part of the core Python language, but learning or using them isn’t compulsory; if they don’t solve any problems that you have, feel free to ignore them. One novel feature of Python's interface as compared to Icon's is that a generator's state is represented as a concrete object (the iterator) that can be passed around to other functions or stored in a data structure.

**See also:**

**PEP 255 - Simple Generators**
Written by Neil Schemenauer, Tim Peters, Magnus Lie Hetland. Implemented mostly by Neil Schemenauer and Tim Peters, with other fixes from the Python Labs crew.
PEP 237: Unifying Long Integers and Integers

In recent versions, the distinction between regular integers, which are 32-bit values on most machines, and long integers, which can be of arbitrary size, was becoming an annoyance. For example, on platforms that support files larger than \(2^{32}\) bytes, the `tell()` method of file objects has to return a long integer. However, there were various bits of Python that expected plain integers and would raise an error if a long integer was provided instead. For example, in Python 1.5, only regular integers could be used as a slice index, and `['abc'][1L:]` would raise a `TypeError` exception with the message 'slice index must be int'.

Python 2.2 will shift values from short to long integers as required. The ‘L’ suffix is no longer needed to indicate a long integer literal, as now the compiler will choose the appropriate type. (Using the ‘L’ suffix will be discouraged in future 2.x versions of Python, triggering a warning in Python 2.4, and probably dropped in Python 3.0.) Many operations that used to raise an `OverflowError` will now return a long integer as their result. For example:

```python
>>> 1234567890123
1234567890123L

>>> 2 ** 64
18446744073709551616L
```

In most cases, integers and long integers will now be treated identically. You can still distinguish them with the `type()` built-in function, but that's rarely needed.

See also:
PEP 237 - Unifying Long Integers and Integers

Written by Moshe Zadka and Guido van Rossum. Implemented mostly by Guido van Rossum.
The most controversial change in Python 2.2 heralds the start of an effort to fix an old design flaw that’s been in Python from the beginning. Currently Python’s division operator, `/`, behaves like C’s division operator when presented with two integer arguments: it returns an integer result that’s truncated down when there would be a fractional part. For example, `3/2` is 1, not 1.5, and `(−1)/2` is -1, not -0.5. This means that the results of division can vary unexpectedly depending on the type of the two operands and because Python is dynamically typed, it can be difficult to determine the possible types of the operands.

(The controversy is over whether this is really a design flaw, and whether it’s worth breaking existing code to fix this. It’s caused endless discussions on python-dev, and in July 2001 erupted into a storm of acidly sarcastic postings on comp.lang.python. I won’t argue for either side here and will stick to describing what’s implemented in 2.2. Read PEP 238 for a summary of arguments and counter-arguments.)

Because this change might break code, it’s being introduced very gradually. Python 2.2 begins the transition, but the switch won’t be complete until Python 3.0.

First, I’ll borrow some terminology from PEP 238. “True division” is the division that most non-programmers are familiar with: `3/2` is 1.5, `1/4` is 0.25, and so forth. “Floor division” is what Python’s `/` operator currently does when given integer operands; the result is the floor of the value returned by true division. “Classic division” is the current mixed behaviour of `/`; it returns the result of floor division when the operands are integers, and returns the result of true division when one of the operands is a floating-point number.
Here are the changes 2.2 introduces:

- A new operator, `//`, is the floor division operator. (Yes, we know it looks like C++’s comment symbol.) `//` always performs floor division no matter what the types of its operands are, so `1 // 2` is 0 and `1.0 // 2.0` is also 0.0.

`//` is always available in Python 2.2; you don’t need to enable it using a `__future__` statement.

- By including a `from __future__ import division` in a module, the `/` operator will be changed to return the result of true division, so `1/2` is 0.5. Without the `__future__` statement, `/` still means classic division. The default meaning of `/` will not change until Python 3.0.

- Classes can define methods called `__truediv__()` and `__floordiv__()` to overload the two division operators. At the C level, there are also slots in the `PyNumberMethods` structure so extension types can define the two operators.

- Python 2.2 supports some command-line arguments for testing whether code will work with the changed division semantics. Running `python` with `-Q warn` will cause a warning to be issued whenever division is applied to two integers. You can use this to find code that’s affected by the change and fix it. By default, Python 2.2 will simply perform classic division without a warning; the warning will be turned on by default in Python 2.3.

See also:

**PEP 238 - Changing the Division Operator**

Written by Moshe Zadka and Guido van Rossum. Implemented by Guido van Rossum..
Unicode Changes

Python’s Unicode support has been enhanced a bit in 2.2. Unicode strings are usually stored as UCS-2, as 16-bit unsigned integers. Python 2.2 can also be compiled to use UCS-4, 32-bit unsigned integers, as its internal encoding by supplying `--enable-unicode=ucs4` to the configure script. (It's also possible to specify `--disable-unicode` to completely disable Unicode support.)

When built to use UCS-4 (a “wide Python”), the interpreter can natively handle Unicode characters from U+000000 to U+110000, so the range of legal values for the `unichr()` function is expanded accordingly. Using an interpreter compiled to use UCS-2 (a “narrow Python”), values greater than 65535 will still cause `unichr()` to raise a `ValueError` exception. This is all described in PEP 261, “Support for ‘wide’ Unicode characters”; consult it for further details.

Another change is simpler to explain. Since their introduction, Unicode strings have supported an `encode()` method to convert the string to a selected encoding such as UTF-8 or Latin-1. A symmetric `decode([*encoding*])()` method has been added to 8-bit strings (though not to Unicode strings) in 2.2. `decode()` assumes that the string is in the specified encoding and decodes it, returning whatever is returned by the codec.

Using this new feature, codecs have been added for tasks not directly related to Unicode. For example, codecs have been added for uu-encoding, MIME’s base64 encoding, and compression with the `zlib` module:

```python
>>> s = """Here is a lengthy piece of redundant, overly verbose ... and repetitive text. ...
... """
```
To convert a class instance to Unicode, a `__unicode__()` method can be defined by a class, analogous to `__str__()`. `encode()`, `decode()`, and `__unicode__()` were implemented by Marc-André Lemburg. The changes to support using UCS-4 internally were implemented by Fredrik Lundh and Martin von Löwis.

**See also:**

**PEP 261 - Support for ‘wide’ Unicode characters**

Written by Paul Prescod.
In Python 2.1, statically nested scopes were added as an optional feature, to be enabled by a `from __future__ import nested_scopes` directive. In 2.2 nested scopes no longer need to be specially enabled, and are now always present. The rest of this section is a copy of the description of nested scopes from my “What’s New in Python 2.1” document; if you read it when 2.1 came out, you can skip the rest of this section.

The largest change introduced in Python 2.1, and made complete in 2.2, is to Python’s scoping rules. In Python 2.0, at any given time there are at most three namespaces used to look up variable names: local, module-level, and the built-in namespace. This often surprised people because it didn’t match their intuitive expectations. For example, a nested recursive function definition doesn’t work:

```python
def f():
    ...
    def g(value):
        ...
        return g(value-1) + 1
    ...
```

The function `g()` will always raise a `NameError` exception, because the binding of the name `g` isn’t in either its local namespace or in the module-level namespace. This isn’t much of a problem in practice (how often do you recursively define interior functions like this?), but this also made using the `lambda` statement clumsier, and this was a problem in practice. In code which uses `lambda` you can often find local variables being copied by passing them as the default values of arguments.

```python
def find(self, name):
    "Return list of any entries equal to 'name'"
```
The readability of Python code written in a strongly functional style suffers greatly as a result.

The most significant change to Python 2.2 is that static scoping has been added to the language to fix this problem. As a first effect, the name=name default argument is now unnecessary in the above example. Put simply, when a given variable name is not assigned a value within a function (by an assignment, or the def, class, or import statements), references to the variable will be looked up in the local namespace of the enclosing scope. A more detailed explanation of the rules, and a dissection of the implementation, can be found in the PEP.

This change may cause some compatibility problems for code where the same variable name is used both at the module level and as a local variable within a function that contains further function definitions. This seems rather unlikely though, since such code would have been pretty confusing to read in the first place.

One side effect of the change is that the from module import * and exec statements have been made illegal inside a function scope under certain conditions. The Python reference manual has said all along that from module import * is only legal at the top level of a module, but the CPython interpreter has never enforced this before. As part of the implementation of nested scopes, the compiler which turns Python source into bytecodes has to generate different code to access variables in a containing scope. from module import * and exec make it impossible for the compiler to figure this out, because they add names to the local namespace that are unknowable at compile time. Therefore, if a function contains function definitions or lambda expressions with free variables, the compiler will flag this by
raising a SyntaxError exception.

To make the preceding explanation a bit clearer, here's an example:

```python
def f():
    # The next line is a syntax error
    exec 'x=2'
    def g():
        return x
```

Line 4 containing the `exec` statement is a syntax error, since `exec` would define a new local variable named `x` whose value should be accessed by `g()`.

This shouldn’t be much of a limitation, since `exec` is rarely used in most Python code (and when it is used, it's often a sign of a poor design anyway).

See also:

**PEP 227 - Statically Nested Scopes**
Written and implemented by Jeremy Hylton.
New and Improved Modules

- The `xmlrpclib` module was contributed to the standard library by Fredrik Lundh, providing support for writing XML-RPC clients. XML-RPC is a simple remote procedure call protocol built on top of HTTP and XML. For example, the following snippet retrieves a list of RSS channels from the O'Reilly Network, and then lists the recent headlines for one channel:

```python
import xmlrpclib
channels = s.meerkat.getChannels()
# channels is a list of dictionaries, like this:
# [{'id': 4, 'title': 'Freshmeat Daily News'},
#  {'id': 190, 'title': '32Bits Online'},
#  {'id': 4549, 'title': '3DGamers'}, ...

# Get the items for one channel
items = s.meerkat.getItems( {'channel': 4} )
# 'items' is another list of dictionaries, like this:
# [{'link': 'http://freshmeat.net/releases/52719/',
#  'description': 'A utility which converts HTML to XSL FO',
#  'title': 'html2fo 0.3 (Default)'}], ...
```

The `SimpleXMLRPCServer` module makes it easy to create straightforward XML-RPC servers. See http://www.xmlrpc.com/ for more information about XML-RPC.

- The new `hmac` module implements the HMAC algorithm described by RFC 2104. (Contributed by Gerhard Häring.)

- Several functions that originally returned lengthy tuples now return pseudo-sequences that still behave like tuples but also have mnemonic attributes such as `memberst_mtime` or `tm_year`. 
The enhanced functions include `stat()`, `fstat()`, `statvfs()`, and `fstatvfs()` in the `os` module, and `localtime()`, `gmtime()`, and `strptime()` in the `time` module.

For example, to obtain a file’s size using the old tuples, you'd end up writing something like `file_size = os.stat(filename)[stat.ST_SIZE]`, but now this can be written more clearly as `file_size = os.stat(filename).st_size`.

The original patch for this feature was contributed by Nick Mathewson.

- The Python profiler has been extensively reworked and various errors in its output have been corrected. (Contributed by Fred L. Drake, Jr. and Tim Peters.)

- The `socket` module can be compiled to support IPv6; specify the `--enable-ipv6` option to Python’s configure script. (Contributed by Jun-ichiro “itojun” Hagino.)

- Two new format characters were added to the `struct` module for 64-bit integers on platforms that support the C `long long` type. \( q \) is for a signed 64-bit integer, and \( Q \) is for an unsigned one. The value is returned in Python’s `long` integer type. (Contributed by Tim Peters.)

- In the interpreter’s interactive mode, there's a new built-in function `help()` that uses the `pydoc` module introduced in Python 2.1 to provide interactive help. `help(object)` displays any available help text about `object`. `help()` with no argument puts you in an online help utility, where you can enter the names of functions, classes, or modules to read their help text. (Contributed by Guido van Rossum, using Ka-Ping Yee’s `pydoc` module.)
Various bugfixes and performance improvements have been made to the SRE engine underlying the re module. For example, the re.sub() and re.split() functions have been rewritten in C. Another contributed patch speeds up certain Unicode character ranges by a factor of two, and a new finditer() method that returns an iterator over all the non-overlapping matches in a given string. (SRE is maintained by Fredrik Lundh. The BIGCHARSET patch was contributed by Martin von Löwis.)

The smtplib module now supports RFC 2487, “Secure SMTP over TLS”, so it’s now possible to encrypt the SMTP traffic between a Python program and the mail transport agent being handed a message. smtplib also supports SMTP authentication. (Contributed by Gerhard Häring.)

The imaplib module, maintained by Piers Lauder, has support for several new extensions: the NAMESPACE extension defined in RFC 2342, SORT, GETACL and SETACL. (Contributed by Anthony Baxter and Michel Pelletier.)

The rfc822 module’s parsing of email addresses is now compliant with RFC 2822, an update to RFC 822. (The module’s name is not going to be changed to rfc2822.) A new package, email, has also been added for parsing and generating e-mail messages. (Contributed by Barry Warsaw, and arising out of his work on Mailman.)

The difflib module now contains a new Differ class for producing human-readable lists of changes (a “delta”) between two sequences of lines of text. There are also two generator functions, ndiff() and restore(), which respectively return a delta from two sequences, or one of the original sequences from a delta. (Grunt work contributed by David Goodger, from ndiff.py
code by Tim Peters who then did the generatorization.)

- New constants `ascii_letters`, `ascii_lowercase`, and `ascii_uppercase` were added to the `string` module. There were several modules in the standard library that used `string.letters` to mean the ranges A-Za-z, but that assumption is incorrect when locales are in use, because `string.letters` varies depending on the set of legal characters defined by the current locale. The buggy modules have all been fixed to use `ascii_letters` instead. (Reported by an unknown person; fixed by Fred L. Drake, Jr.)

- The `mimetypes` module now makes it easier to use alternative MIME-type databases by the addition of a `MimeTypes` class, which takes a list of filenames to be parsed. (Contributed by Fred L. Drake, Jr.)

- A `Timer` class was added to the `threading` module that allows scheduling an activity to happen at some future time. (Contributed by Itamar Shtull-Trauring.)
Interpret Changes and Fixes

Some of the changes only affect people who deal with the Python interpreter at the C level because they're writing Python extension modules, embedding the interpreter, or just hacking on the interpreter itself. If you only write Python code, none of the changes described here will affect you very much.

- Profiling and tracing functions can now be implemented in C, which can operate at much higher speeds than Python-based functions and should reduce the overhead of profiling and tracing. This will be of interest to authors of development environments for Python. Two new C functions were added to Python’s API, `PyEval_SetProfile()` and `PyEval_SetTrace()`. The existing `sys.setprofile()` and `sys.settrace()` functions still exist, and have simply been changed to use the new C-level interface. (Contributed by Fred L. Drake, Jr.)

- Another low-level API, primarily of interest to implementors of Python debuggers and development tools, was added. `PyInterpreterState_Head()` and `PyInterpreterState_Next()` let a caller walk through all the existing interpreter objects; `PyInterpreterState_ThreadHead()` and `PyThreadState_Next()` allow looping over all the thread states for a given interpreter. (Contributed by David Beazley.)

- The C-level interface to the garbage collector has been changed to make it easier to write extension types that support garbage collection and to debug misuses of the functions. Various functions have slightly different semantics, so a bunch of functions had to be renamed. Extensions that use the old API will still compile but will not participate in garbage collection, so updating them for 2.2 should be considered fairly high priority.
To upgrade an extension module to the new API, perform the following steps:

- Rename `Py_TPFLAGS_GC()` to `PyTPFLAGS_HAVE_GC()`.

- Use `PyObject_GC_New()` or `PyObject_GC_NewVar()` to allocate objects, and `PyObject_GC_Del()` to deallocate them.

- Rename `PyObject_GC_Init()` to `PyObject_GC_Track()` and `PyObject_GC_Fini()` to `PyObject_GC_UnTrack()`.

- Remove `PyGC_HEAD_SIZE()` from object size calculations.

- Remove calls to `PyObject_AS_GC()` and `PyObject_FROM_GC()`.

- A new `et` format sequence was added to `PyArg_ParseTuple()`. `et` takes both a parameter and an encoding name, and converts the parameter to the given encoding if the parameter turns out to be a Unicode string, or leaves it alone if it's an 8-bit string, assuming it to already be in the desired encoding. This differs from the `es` format character, which assumes that 8-bit strings are in Python’s default ASCII encoding and converts them to the specified new encoding. (Contributed by M.-A. Lemburg, and used for the MBCS support on Windows described in the following section.)

- A different argument parsing function, `PyArg_UnpackTuple()`, has been added that’s simpler and presumably faster. Instead of specifying a format string, the caller simply gives the minimum and maximum number of arguments expected, and a set of pointers to `PyObject*` variables that will be filled in with argument values.

- Two new flags `METH_NOARGS` and `METH_O` are available in method
definition tables to simplify implementation of methods with no arguments or a single untyped argument. Calling such methods is more efficient than calling a corresponding method that uses METH_VARARGS. Also, the old METH_OLDARGS style of writing C methods is now officially deprecated.

- Two new wrapper functions, PyOS_snprintf() and PyOS_vsnprintf() were added to provide cross-platform implementations for the relatively new snprintf() and vsnprintf() C lib APIs. In contrast to the standard sprintf() and vsprintf() functions, the Python versions check the bounds of the buffer used to protect against buffer overruns. (Contributed by M.-A. Lemburg.)

- The _PyTuple_Resize() function has lost an unused parameter, so now it takes 2 parameters instead of 3. The third argument was never used, and can simply be discarded when porting code from earlier versions to Python 2.2.
Other Changes and Fixes

As usual there were a bunch of other improvements and bugfixes scattered throughout the source tree. A search through the CVS change logs finds there were 527 patches applied and 683 bugs fixed between Python 2.1 and 2.2; 2.2.1 applied 139 patches and fixed 143 bugs; 2.2.2 applied 106 patches and fixed 82 bugs. These figures are likely to be underestimates.

Some of the more notable changes are:

- The code for the MacOS port for Python, maintained by Jack Jansen, is now kept in the main Python CVS tree, and many changes have been made to support MacOS X.

The most significant change is the ability to build Python as a framework, enabled by supplying the `--enable-framework` option to the configure script when compiling Python. According to Jack Jansen, “This installs a self- contained Python installation plus the OS X framework “glue” into `/Library/Frameworks/Python.framework` (or another location of choice). For now there is little immediate added benefit to this (actually, there is the disadvantage that you have to change your PATH to be able to find Python), but it is the basis for creating a full-blown Python application, porting the MacPython IDE, possibly using Python as a standard OSA scripting language and much more.”

Most of the MacPython toolbox modules, which interface to MacOS APIs such as windowing, QuickTime, scripting, etc. have been ported to OS X, but they've been left commented out in `setup.py`. People who want to experiment with these modules can uncomment them manually.
- Keyword arguments passed to built-in functions that don’t take them now cause a `TypeError` exception to be raised, with the message “`function` takes no keyword arguments”.

- Weak references, added in Python 2.1 as an extension module, are now part of the core because they’re used in the implementation of new-style classes. The `ReferenceError` exception has therefore moved from the `weakref` module to become a built-in exception.

- A new script, `Tools/scripts/cleanfuture.py` by Tim Peters, automatically removes obsolete `__future__` statements from Python source code.

- An additional `flags` argument has been added to the built-in function `compile()`, so the behaviour of `__future__` statements can now be correctly observed in simulated shells, such as those presented by IDLE and other development environments. This is described in PEP 264. (Contributed by Michael Hudson.)

- The new license introduced with Python 1.6 wasn’t GPL-compatible. This is fixed by some minor textual changes to the 2.2 license, so it’s now legal to embed Python inside a GPLed program again. Note that Python itself is not GPLed, but instead is under a license that’s essentially equivalent to the BSD license, same as it always was. The license changes were also applied to the Python 2.0.1 and 2.1.1 releases.

- When presented with a Unicode filename on Windows, Python will now convert it to an MBCS encoded string, as used by the Microsoft file APIs. As MBCS is explicitly used by the file APIs, Python’s choice of ASCII as the default encoding turns out to be an annoyance. On Unix, the locale’s character set is used if `locale.nl_langinfo(CODESET)()` is available. (Windows support
was contributed by Mark Hammond with assistance from Marc-André Lemburg. Unix support was added by Martin von Löwis.)

- Large file support is now enabled on Windows. (Contributed by Tim Peters.)

- The Tools/scripts/ftpmirror.py script now parses a .netrc file, if you have one. (Contributed by Mike Romberg.)

- Some features of the object returned by the `xrange()` function are now deprecated, and trigger warnings when they're accessed; they'll disappear in Python 2.3. `xrange` objects tried to pretend they were full sequence types by supporting slicing, sequence multiplication, and the `in` operator, but these features were rarely used and therefore buggy. The `tolist()` method and the `start`, `stop`, and `step` attributes are also being deprecated. At the C level, the fourth argument to the `PyRange_New()` function, `repeat`, has also been deprecated.

- There were a bunch of patches to the dictionary implementation, mostly to fix potential core dumps if a dictionary contains objects that sneakily changed their hash value, or mutated the dictionary they were contained in. For a while python-dev fell into a gentle rhythm of Michael Hudson finding a case that dumped core, Tim Peters fixing the bug, Michael finding another case, and round and round it went.

- On Windows, Python can now be compiled with Borland C thanks to a number of patches contributed by Stephen Hansen, though the result isn't fully functional yet. (But this is progress...)

- Another Windows enhancement: Wise Solutions generously offered PythonLabs use of their InstallerMaster 8.1 system. Earlier PythonLabs Windows installers used Wise 5.0a, which was beginning to show its age. (Packaged up by Tim Peters.)
Files ending in `.pyw` can now be imported on Windows. `.pyw` is a Windows-only thing, used to indicate that a script needs to be run using PYTHONW.EXE instead of PYTHON.EXE in order to prevent a DOS console from popping up to display the output. This patch makes it possible to import such scripts, in case they're also usable as modules. (Implemented by David Bolen.)

On platforms where Python uses the C `dlopen()` function to load extension modules, it's now possible to set the flags used by `dlopen()` using the `sys.getdlopenflags()` and `sys.setdlopenflags()` functions. (Contributed by Bram Stolk.)

The `pow()` built-in function no longer supports 3 arguments when floating-point numbers are supplied. `pow(x, y, z)` returns `(x**y) % z`, but this is never useful for floating point numbers, and the final result varies unpredictably depending on the platform. A call such as `pow(2.0, 8.0, 7.0)` will now raise a `TypeError` exception.
Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Fred Bremmer, Keith Briggs, Andrew Dalke, Fred L. Drake, Jr., Carel Fellinger, David Goodger, Mark Hammond, Stephen Hansen, Michael Hudson, Jack Jansen, Marc-André Lemburg, Martin von Löwis, Fredrik Lundh, Michael McLay, Nick Mathewson, Paul Moore, Gustavo Niemeyer, Don O'Donnell, Joonas Paalasma, Tim Peters, Jens Quade, Tom Reinhardt, Neil Schemenauer, Guido van Rossum, Greg Ward, Edward Welbourne.
What’s New in Python 2.1

Author: A.M. Kuchling
Introduction

This article explains the new features in Python 2.1. While there aren’t as many changes in 2.1 as there were in Python 2.0, there are still some pleasant surprises in store. 2.1 is the first release to be steered through the use of Python Enhancement Proposals, or PEPs, so most of the sizable changes have accompanying PEPs that provide more complete documentation and a design rationale for the change. This article doesn’t attempt to document the new features completely, but simply provides an overview of the new features for Python programmers. Refer to the Python 2.1 documentation, or to the specific PEP, for more details about any new feature that particularly interests you.

One recent goal of the Python development team has been to accelerate the pace of new releases, with a new release coming every 6 to 9 months. 2.1 is the first release to come out at this faster pace, with the first alpha appearing in January, 3 months after the final version of 2.0 was released.

The final release of Python 2.1 was made on April 17, 2001.
The largest change in Python 2.1 is to Python’s scoping rules. In Python 2.0, at any given time there are at most three namespaces used to look up variable names: local, module-level, and the built-in namespace. This often surprised people because it didn’t match their intuitive expectations. For example, a nested recursive function definition doesn’t work:

```python
def f():
    ...
    def g(value):
        ...
        return g(value-1) + 1
    ...
```

The function `g()` will always raise a `NameError` exception, because the binding of the name `g` isn’t in either its local namespace or in the module-level namespace. This isn’t much of a problem in practice (how often do you recursively define interior functions like this?), but this also made using the `lambda` statement clumsier, and this was a problem in practice. In code which uses `lambda` you can often find local variables being copied by passing them as the default values of arguments.

```python
def find(self, name):
    "Return list of any entries equal to 'name'"
    L = filter(lambda x, name=name: x == name,
               self.list_attribute)
    return L
```

The readability of Python code written in a strongly functional style suffers greatly as a result.

The most significant change to Python 2.1 is that static scoping has been added to the language to fix this problem. As a first effect, the
default argument is now unnecessary in the above example. Put simply, when a given variable name is not assigned a value within a function (by an assignment, or the \texttt{def}, \texttt{class}, or \texttt{import} statements), references to the variable will be looked up in the local namespace of the enclosing scope. A more detailed explanation of the rules, and a dissection of the implementation, can be found in the PEP.

This change may cause some compatibility problems for code where the same variable name is used both at the module level and as a local variable within a function that contains further function definitions. This seems rather unlikely though, since such code would have been pretty confusing to read in the first place.

One side effect of the change is that the \texttt{from module import *} and \texttt{exec} statements have been made illegal inside a function scope under certain conditions. The Python reference manual has said all along that \texttt{from module import *} is only legal at the top level of a module, but the CPython interpreter has never enforced this before. As part of the implementation of nested scopes, the compiler which turns Python source into bytecodes has to generate different code to access variables in a containing scope. \texttt{from module import *} and \texttt{exec} make it impossible for the compiler to figure this out, because they add names to the local namespace that are unknowable at compile time. Therefore, if a function contains function definitions or \texttt{lambda} expressions with free variables, the compiler will flag this by raising a \texttt{SyntaxError} exception.

To make the preceding explanation a bit clearer, here's an example:

```python
x = 1
def f():
    # The next line is a syntax error
    exec 'x=2'
    def g():
```
Line 4 containing the `exec` statement is a syntax error, since `exec` would define a new local variable named `x` whose value should be accessed by `g()`.

This shouldn’t be much of a limitation, since `exec` is rarely used in most Python code (and when it is used, it’s often a sign of a poor design anyway).

Compatibility concerns have led to nested scopes being introduced gradually; in Python 2.1, they aren’t enabled by default, but can be turned on within a module by using a future statement as described in PEP 236. (See the following section for further discussion of PEP 236.) In Python 2.2, nested scopes will become the default and there will be no way to turn them off, but users will have had all of 2.1’s lifetime to fix any breakage resulting from their introduction.

See also:

**PEP 227 - Statically Nested Scopes**

Written and implemented by Jeremy Hylton.
PEP 236: __future__ Directives

The reaction to nested scopes was widespread concern about the dangers of breaking code with the 2.1 release, and it was strong enough to make the Pythoneers take a more conservative approach. This approach consists of introducing a convention for enabling optional functionality in release N that will become compulsory in release N+1.

The syntax uses a \texttt{from...import} statement using the reserved module name \texttt{__future__}. Nested scopes can be enabled by the following statement:

\begin{lstlisting}
from __future__ import nested_scopes
\end{lstlisting}

While it looks like a normal \texttt{import} statement, it's not; there are strict rules on where such a future statement can be put. They can only be at the top of a module, and must precede any Python code or regular \texttt{import} statements. This is because such statements can affect how the Python bytecode compiler parses code and generates bytecode, so they must precede any statement that will result in bytecodes being produced.

See also:

PEP 236 - Back to the __future__

Written by Tim Peters, and primarily implemented by Jeremy Hylton.
PEP 207: Rich Comparisons

In earlier versions, Python’s support for implementing comparisons on user-defined classes and extension types was quite simple. Classes could implement a `__cmp__()` method that was given two instances of a class, and could only return 0 if they were equal or +1 or -1 if they weren’t; the method couldn’t raise an exception or return anything other than a Boolean value. Users of Numeric Python often found this model too weak and restrictive, because in the number-crunching programs that numeric Python is used for, it would be more useful to be able to perform elementwise comparisons of two matrices, returning a matrix containing the results of a given comparison for each element. If the two matrices are of different sizes, then the compare has to be able to raise an exception to signal the error.

In Python 2.1, rich comparisons were added in order to support this need. Python classes can now individually overload each of the `<`, `<=`, `>`, `>=`, `==`, and `!=` operations. The new magic method names are:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Method name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td><code>__lt__()</code></td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td><code>__le__()</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td><code>__gt__()</code></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td><code>__ge__()</code></td>
</tr>
<tr>
<td><code>==</code></td>
<td><code>__eq__()</code></td>
</tr>
<tr>
<td><code>!=</code></td>
<td><code>__ne__()</code></td>
</tr>
</tbody>
</table>

(The magic methods are named after the corresponding Fortran operators `.LT.` `.LE.` &c. Numeric programmers are almost certainly quite familiar with these names and will find them easy to remember.)

Each of these magic methods is of the form `method(self, other),`
where `self` will be the object on the left-hand side of the operator, while `other` will be the object on the right-hand side. For example, the expression `A < B` will cause `A.__lt__(B)` to be called.

Each of these magic methods can return anything at all: a Boolean, a matrix, a list, or any other Python object. Alternatively they can raise an exception if the comparison is impossible, inconsistent, or otherwise meaningless.

The built-in `cmp(A,B)()` function can use the rich comparison machinery, and now accepts an optional argument specifying which comparison operation to use; this is given as one of the strings `"<"`, `"<="`, `">"`, `">="`, `"=="`, or `"!="`. If called without the optional third argument, `cmp()` will only return -1, 0, or +1 as in previous versions of Python; otherwise it will call the appropriate method and can return any Python object.

There are also corresponding changes of interest to C programmers; there’s a new slot `tp_richcmp` in type objects and an API for performing a given rich comparison. I won’t cover the C API here, but will refer you to PEP 207, or to 2.1’s C API documentation, for the full list of related functions.

**See also:**

**PEP 207 - Rich Comparisons**

Written by Guido van Rossum, heavily based on earlier work by David Ascher, and implemented by Guido van Rossum.
Over its 10 years of existence, Python has accumulated a certain number of obsolete modules and features along the way. It’s difficult to know when a feature is safe to remove, since there’s no way of knowing how much code uses it — perhaps no programs depend on the feature, or perhaps many do. To enable removing old features in a more structured way, a warning framework was added. When the Python developers want to get rid of a feature, it will first trigger a warning in the next version of Python. The following Python version can then drop the feature, and users will have had a full release cycle to remove uses of the old feature.

Python 2.1 adds the warning framework to be used in this scheme. It adds a `warnings` module that provide functions to issue warnings, and to filter out warnings that you don’t want to be displayed. Third-party modules can also use this framework to deprecate old features that they no longer wish to support.

For example, in Python 2.1 the `regex` module is deprecated, so importing it causes a warning to be printed:

```python
>>> import regex
__main__:1: DeprecationWarning: the regex module is deprecated; please use the re module
```

Warnings can be issued by calling the `warnings.warn()` function:

```python
warnings.warn("feature X no longer supported")
```

The first parameter is the warning message; an additional optional parameters can be used to specify a particular warning category.

Filters can be added to disable certain warnings; a regular
expression pattern can be applied to the message or to the module name in order to suppress a warning. For example, you may have a program that uses the regex module and not want to spare the time to convert it to use the re module right now. The warning can be suppressed by calling

```python
import warnings
warnings.filterwarnings(action = 'ignore',
    message='.*regex module is deprecated',
    category=DeprecationWarning,
    module = '__main__')
```

This adds a filter that will apply only to warnings of the class DeprecationWarning triggered in the __main__ module, and applies a regular expression to only match the message about the regex module being deprecated, and will cause such warnings to be ignored. Warnings can also be printed only once, printed every time the offending code is executed, or turned into exceptions that will cause the program to stop (unless the exceptions are caught in the usual way, of course).

Functions were also added to Python’s C API for issuing warnings; refer to PEP 230 or to Python’s API documentation for the details.

See also:

**PEP 5 - Guidelines for Language Evolution**

Written by Paul Prescod, to specify procedures to be followed when removing old features from Python. The policy described in this PEP hasn’t been officially adopted, but the eventual policy probably won’t be too different from Prescod’s proposal.

**PEP 230 - Warning Framework**

Written and implemented by Guido van Rossum.
PEP 229: New Build System

When compiling Python, the user had to go in and edit the Modules/Setup file in order to enable various additional modules; the default set is relatively small and limited to modules that compile on most Unix platforms. This means that on Unix platforms with many more features, most notably Linux, Python installations often don’t contain all useful modules they could.

Python 2.0 added the Distutils, a set of modules for distributing and installing extensions. In Python 2.1, the Distutils are used to compile much of the standard library of extension modules, autodetecting which ones are supported on the current machine. It’s hoped that this will make Python installations easier and more featureful.

Instead of having to edit the Modules/Setup file in order to enable modules, a setup.py script in the top directory of the Python source distribution is run at build time, and attempts to discover which modules can be enabled by examining the modules and header files on the system. If a module is configured in Modules/Setup, the setup.py script won’t attempt to compile that module and will defer to the Modules/Setup file’s contents. This provides a way to specify any strange command-line flags or libraries that are required for a specific platform.

In another far-reaching change to the build mechanism, Neil Schemenauer restructured things so Python now uses a single makefile that isn’t recursive, instead of makefiles in the top directory and in each of the Python/, Parser/, Objects/, and Modules/ subdirectories. This makes building Python faster and also makes hacking the Makefiles clearer and simpler.

See also:
PEP 229 - Using Distutils to Build Python

Written and implemented by A.M. Kuchling.
PEP 205: Weak References

Weak references, available through the `weakref` module, are a minor but useful new data type in the Python programmer’s toolbox.

Storing a reference to an object (say, in a dictionary or a list) has the side effect of keeping that object alive forever. There are a few specific cases where this behaviour is undesirable; object caches being the most common one, and another being circular references in data structures such as trees.

For example, consider a memoizing function that caches the results of another function `f(x)()` by storing the function’s argument and its result in a dictionary:

```python
_cache = {}
def memoize(x):
    if _cache.has_key(x):
        return _cache[x]

retval = f(x)

# Cache the returned object
_cache[x] = retval

return retval
```

This version works for simple things such as integers, but it has a side effect; the `_cache` dictionary holds a reference to the return values, so they’ll never be deallocated until the Python process exits and cleans up. This isn’t very noticeable for integers, but if `f()` returns an object, or a data structure that takes up a lot of memory, this can be a problem.

Weak references provide a way to implement a cache that won’t keep objects alive beyond their time. If an object is only accessible
through weak references, the object will be deallocated and the weak references will now indicate that the object it referred to no longer exists. A weak reference to an object `obj` is created by calling `wr = weakref.ref(obj)`. The object being referred to is returned by calling the weak reference as if it were a function: `wr()`. It will return the referenced object, or `None` if the object no longer exists.

This makes it possible to write a `memoize()` function whose cache doesn’t keep objects alive, by storing weak references in the cache.

```
_cache = {}
def memoize(x):
    if _cache.has_key(x):
        obj = _cache[x]()
        # If weak reference object still exists,
        # return it
        if obj is not None: return obj

    retval = f(x)
    # Cache a weak reference
    _cache[x] = weakref.ref(retval)
    return retval
```

The `weakref` module also allows creating proxy objects which behave like weak references — an object referenced only by proxy objects is deallocated – but instead of requiring an explicit call to retrieve the object, the proxy transparently forwards all operations to the object as long as the object still exists. If the object is deallocated, attempting to use a proxy will cause a `weakref.ReferenceError` exception to be raised.

```
proxy = weakref.proxy(obj)
proxy.attr       # Equivalent to obj.attr
proxy.meth()     # Equivalent to obj.meth()
del obj
proxy.attr       # raises weakref.ReferenceError
```
See also:

**PEP 205 - Weak References**
Written and implemented by Fred L. Drake, Jr.
In Python 2.1, functions can now have arbitrary information attached to them. People were often using docstrings to hold information about functions and methods, because the __doc__ attribute was the only way of attaching any information to a function. For example, in the Zope Web application server, functions are marked as safe for public access by having a docstring, and in John Aycock's SPARK parsing framework, docstrings hold parts of the BNF grammar to be parsed. This overloading is unfortunate, since docstrings are really intended to hold a function's documentation; for example, it means you can't properly document functions intended for private use in Zope.

Arbitrary attributes can now be set and retrieved on functions using the regular Python syntax:

```python
def f(): pass
f.publish = 1
f.secure = 1
f.grammar = "A ::= B (C D)"
```

The dictionary containing attributes can be accessed as the function's __dict__. Unlike the __dict__ attribute of class instances, in functions you can actually assign a new dictionary to __dict__, though the new value is restricted to a regular Python dictionary; you can't be tricky and set it to a UserDict instance, or any other random object that behaves like a mapping.

See also:

**PEP 232 - Function Attributes**

Written and implemented by Barry Warsaw.
Some operating systems have filesystems that are case-insensitive, MacOS and Windows being the primary examples; on these systems, it's impossible to distinguish the filenames `FILE.PY` and `file.py`, even though they do store the file's name in its original case (they're case-preserving, too).

In Python 2.1, the `import` statement will work to simulate case-sensitivity on case-insensitive platforms. Python will now search for the first case-sensitive match by default, raising an `ImportError` if no such file is found, so `import file` will not import a module named `FILE.PY`. Case-insensitive matching can be requested by setting the `PYTHONCASEOK` environment variable before starting the Python interpreter.
When using the Python interpreter interactively, the output of commands is displayed using the built-in `repr()` function. In Python 2.1, the variable `sys.displayhook()` can be set to a callable object which will be called instead of `repr()`. For example, you can set it to a special pretty-printing function:

```python
>>> # Create a recursive data structure
... L = [1,2,3]
>>> L.append(L)
>>> L # Show Python's default output
[1, 2, 3, [...]]
>>> # Use pprint.pprint() as the display function
... import sys, pprint
>>> sys.displayhook = pprint.pprint
>>> L
[1, 2, 3, <Recursion on list with id=135143996>]
```
How numeric coercion is done at the C level was significantly modified. This will only affect the authors of C extensions to Python, allowing them more flexibility in writing extension types that support numeric operations.

Extension types can now set the type flag `Py_TPFLAGS_CHECKTYPES` in their `PyTypeObject` structure to indicate that they support the new coercion model. In such extension types, the numeric slot functions can no longer assume that they’ll be passed two arguments of the same type; instead they may be passed two arguments of differing types, and can then perform their own internal coercion. If the slot function is passed a type it can't handle, it can indicate the failure by returning a reference to the `Py_NotImplemented` singleton value. The numeric functions of the other type will then be tried, and perhaps they can handle the operation; if the other type also returns `Py_NotImplemented`, then a `TypeError` will be raised. Numeric methods written in Python can also return `Py_NotImplemented`, causing the interpreter to act as if the method did not exist (perhaps raising a `TypeError`, perhaps trying another object's numeric methods).

See also:

**PEP 208 - Reworking the Coercion Model**

Written and implemented by Neil Schemenauer, heavily based upon earlier work by Marc-André Lemburg. Read this to understand the fine points of how numeric operations will now be processed at the C level.
PEP 241: Metadata in Python Packages

A common complaint from Python users is that there’s no single catalog of all the Python modules in existence. T. Middleton’s Vaults of Parnassus at http://www.vex.net/parnassus/ are the largest catalog of Python modules, but registering software at the Vaults is optional, and many people don’t bother.

As a first small step toward fixing the problem, Python software packaged using the Distutils `sdist` command will include a file named `PKG-INFO` containing information about the package such as its name, version, and author (metadata, in cataloguing terminology). PEP 241 contains the full list of fields that can be present in the `PKG-INFO` file. As people began to package their software using Python 2.1, more and more packages will include metadata, making it possible to build automated cataloguing systems and experiment with them. With the result experience, perhaps it’ll be possible to design a really good catalog and then build support for it into Python 2.2. For example, the Distutils `sdist` and `bdist_*` commands could support an `upload` option that would automatically upload your package to a catalog server.

You can start creating packages containing `PKG-INFO` even if you’re not using Python 2.1, since a new release of the Distutils will be made for users of earlier Python versions. Version 1.0.2 of the Distutils includes the changes described in PEP 241, as well as various bugfixes and enhancements. It will be available from the Distutils SIG at https://www.python.org/community/sigs/current/distutils-sig/.

See also:

PEP 241 - Metadata for Python Software Packages
Written and implemented by A.M. Kuchling.

**PEP 243 - Module Repository Upload Mechanism**

Written by Sean Reifschneider, this draft PEP describes a proposed mechanism for uploading Python packages to a central server.
New and Improved Modules

- Ka-Ping Yee contributed two new modules: `inspect.py`, a module for getting information about live Python code, and `pydoc.py`, a module for interactively converting docstrings to HTML or text. As a bonus, Tools/scripts/pydoc, which is now automatically installed, uses `pydoc.py` to display documentation given a Python module, package, or class name. For example, `pydoc xml.dom` displays the following:

```
Python Library Documentation: package xml.dom in xml

NAME
  xml.dom - W3C Document Object Model implementation for

FILE
  /usr/local/lib/python2.1/xml/dom/__init__.pyc

DESCRIPTION
  The Python mapping of the Document Object Model is documented in the section on the xml.dom
  Python Library Reference in the section on the xml.dom

  This package contains the following modules:
  ...
```

`pydoc` also includes a Tk-based interactive help browser. `pydoc` quickly becomes addictive; try it out!

- Two different modules for unit testing were added to the standard library. The `doctest` module, contributed by Tim Peters, provides a testing framework based on running embedded examples in docstrings and comparing the results against the expected output. PyUnit, contributed by Steve Purcell, is a unit testing framework inspired by JUnit, which was in turn an adaptation of Kent Beck's Smalltalk testing
framework. See http://pyunit.sourceforge.net/ for more information about PyUnit.

- The difflib module contains a class, SequenceMatcher, which compares two sequences and computes the changes required to transform one sequence into the other. For example, this module can be used to write a tool similar to the Unix diff program, and in fact the sample program Tools/scripts/ndiff.py demonstrates how to write such a script.

- curses.panel, a wrapper for the panel library, part of ncurses and of SYSV curses, was contributed by Thomas Gellekum. The panel library provides windows with the additional feature of depth. Windows can be moved higher or lower in the depth ordering, and the panel library figures out where panels overlap and which sections are visible.

- The PyXML package has gone through a few releases since Python 2.0, and Python 2.1 includes an updated version of the xml package. Some of the noteworthy changes include support for Expat 1.2 and later versions, the ability for Expat parsers to handle files in any encoding supported by Python, and various bugfixes for SAX, DOM, and the minidom module.

- Ping also contributed another hook for handling uncaught exceptions. sys.excepthook() can be set to a callable object. When an exception isn’t caught by any try...except blocks, the exception will be passed to sys.excepthook(), which can then do whatever it likes. At the Ninth Python Conference, Ping demonstrated an application for this hook: printing an extended traceback that not only lists the stack frames, but also lists the function arguments and the local variables for each frame.
Various functions in the `time` module, such as `asctime()` and `localtime()`, require a floating point argument containing the time in seconds since the epoch. The most common use of these functions is to work with the current time, so the floating point argument has been made optional; when a value isn’t provided, the current time will be used. For example, log file entries usually need a string containing the current time; in Python 2.1, `time.asctime()` can be used, instead of the lengthier `time.asctime(time.localtime(time.time()))` that was previously required.

This change was proposed and implemented by Thomas Wouters.

The `ftplib` module now defaults to retrieving files in passive mode, because passive mode is more likely to work from behind a firewall. This request came from the Debian bug tracking system, since other Debian packages use `ftplib` to retrieve files and then don’t work from behind a firewall. It’s deemed unlikely that this will cause problems for anyone, because Netscape defaults to passive mode and few people complain, but if passive mode is unsuitable for your application or network setup, call `set_pasv(0)` on FTP objects to disable passive mode.

Support for raw socket access has been added to the `socket` module, contributed by Grant Edwards.

The `pstats` module now contains a simple interactive statistics browser for displaying timing profiles for Python programs, invoked when the module is run as a script. Contributed by Eric S. Raymond.

A new implementation-dependent function,
sys._getframe([depth])(), has been added to return a given frame object from the current call stack. sys._getframe() returns the frame at the top of the call stack; if the optional integer argument depth is supplied, the function returns the frame that is depth calls below the top of the stack. For example, sys._getframe(1) returns the caller’s frame object.

This function is only present in CPython, not in Jython or the .NET implementation. Use it for debugging, and resist the temptation to put it into production code.
There were relatively few smaller changes made in Python 2.1 due to the shorter release cycle. A search through the CVS change logs turns up 117 patches applied, and 136 bugs fixed; both figures are likely to be underestimates. Some of the more notable changes are:

- A specialized object allocator is now optionally available, that should be faster than the system `malloc()` and have less memory overhead. The allocator uses C’s `malloc()` function to get large pools of memory, and then fulfills smaller memory requests from these pools. It can be enabled by providing the `--with-pymalloc` option to the `configure` script; see `Objects/obmalloc.c` for the implementation details.

Authors of C extension modules should test their code with the object allocator enabled, because some incorrect code may break, causing core dumps at runtime. There are a bunch of memory allocation functions in Python's C API that have previously been just aliases for the C library’s `malloc()` and `free()`, meaning that if you accidentally called mismatched functions, the error wouldn’t be noticeable. When the object allocator is enabled, these functions aren’t aliases of `malloc()` and `free()` any more, and calling the wrong function to free memory will get you a core dump. For example, if memory was allocated using `PyMem_New()`, it has to be freed using `PyMem_Del()`, not `free()`. A few modules included with Python fell afoul of this and had to be fixed; doubtless there are more third-party modules that will have the same problem.

The object allocator was contributed by Vladimir Marangozov.
The speed of line-oriented file I/O has been improved because people often complain about its lack of speed, and because it's often been used as a naïve benchmark. The `readline()` method of file objects has therefore been rewritten to be much faster. The exact amount of the speedup will vary from platform to platform depending on how slow the C library’s `getc()` was, but is around 66%, and potentially much faster on some particular operating systems. Tim Peters did much of the benchmarking and coding for this change, motivated by a discussion in comp.lang.python.

A new module and method for file objects was also added, contributed by Jeff Epler. The new method, `xreadlines()`, is similar to the existing `xrange()` built-in. `xreadlines()` returns an opaque sequence object that only supports being iterated over, reading a line on every iteration but not reading the entire file into memory as the existing `readlines()` method does. You’d use it like this:

```python
for line in sys.stdin.xreadlines():
    # ... do something for each line ...
...
```

For a fuller discussion of the line I/O changes, see the python-dev summary for January 1–15, 2001 at https://mail.python.org/pipermail/python-dev/2001-January/.

A new method, `popitem()`, was added to dictionaries to enable destructively iterating through the contents of a dictionary; this can be faster for large dictionaries because there’s no need to construct a list containing all the keys or values. `D.popitem()` removes a random `(key, value)` pair from the dictionary `D` and returns it as a 2-tuple. This was implemented mostly by Tim Peters and Guido van Rossum, after a suggestion and
preliminary patch by Moshe Zadka.

- Modules can now control which names are imported when `from module import *` is used, by defining an `__all__` attribute containing a list of names that will be imported. One common complaint is that if the module imports other modules such as `sys` or `string`, `from module import *` will add them to the importing module’s namespace. To fix this, simply list the public names in `__all__`:

  ```python
  # List public names
  __all__ = ['Database', 'open']
  ```

  A stricter version of this patch was first suggested and implemented by Ben Wolfson, but after some python-dev discussion, a weaker final version was checked in.

- Applying `repr()` to strings previously used octal escapes for non-printable characters; for example, a newline was `'\012'`. This was a vestigial trace of Python’s C ancestry, but today octal is of very little practical use. Ka-Ping Yee suggested using hex escapes instead of octal ones, and using the `\n`, `\t`, `\r` escapes for the appropriate characters, and implemented this new formatting.

- Syntax errors detected at compile-time can now raise exceptions containing the filename and line number of the error, a pleasant side effect of the compiler reorganization done by Jeremy Hylton.

- C extensions which import other modules have been changed to use `PyImport_ImportModule()`, which means that they will use any import hooks that have been installed. This is also encouraged for third-party extensions that need to import some other module from C code.
The size of the Unicode character database was shrunk by another 340K thanks to Fredrik Lundh.

Some new ports were contributed: MacOS X (by Steven Majewski), Cygwin (by Jason Tishler); RISCOS (by Dietmar Schwertberger); Unixware 7 (by Billy G. Allie).

And there's the usual list of minor bugfixes, minor memory leaks, docstring edits, and other tweaks, too lengthy to be worth itemizing; see the CVS logs for the full details if you want them.
Acknowledgements

The author would like to thank the following people for offering suggestions on various drafts of this article: Graeme Cross, David Goodger, Jay Graves, Michael Hudson, Marc-André Lemburg, Fredrik Lundh, Neil Schemenauer, Thomas Wouters.
What’s New in Python 2.0

Author: A.M. Kuchling and Moshe Zadka
A new release of Python, version 2.0, was released on October 16, 2000. This article covers the exciting new features in 2.0, highlights some other useful changes, and points out a few incompatible changes that may require rewriting code.

Python's development never completely stops between releases, and a steady flow of bug fixes and improvements are always being submitted. A host of minor fixes, a few optimizations, additional docstrings, and better error messages went into 2.0; to list them all would be impossible, but they're certainly significant. Consult the publicly-available CVS logs if you want to see the full list. This progress is due to the five developers working for PythonLabs are now getting paid to spend their days fixing bugs, and also due to the improved communication resulting from moving to SourceForge.
What About Python 1.6?

Python 1.6 can be thought of as the Contractual Obligations Python release. After the core development team left CNRI in May 2000, CNRI requested that a 1.6 release be created, containing all the work on Python that had been performed at CNRI. Python 1.6 therefore represents the state of the CVS tree as of May 2000, with the most significant new feature being Unicode support. Development continued after May, of course, so the 1.6 tree received a few fixes to ensure that it’s forward-compatible with Python 2.0. 1.6 is therefore part of Python’s evolution, and not a side branch.

So, should you take much interest in Python 1.6? Probably not. The 1.6final and 2.0beta1 releases were made on the same day (September 5, 2000), the plan being to finalize Python 2.0 within a month or so. If you have applications to maintain, there seems little point in breaking things by moving to 1.6, fixing them, and then having another round of breakage within a month by moving to 2.0; you’re better off just going straight to 2.0. Most of the really interesting features described in this document are only in 2.0, because a lot of work was done between May and September.
New Development Process

The most important change in Python 2.0 may not be to the code at all, but to how Python is developed: in May 2000 the Python developers began using the tools made available by SourceForge for storing source code, tracking bug reports, and managing the queue of patch submissions. To report bugs or submit patches for Python 2.0, use the bug tracking and patch manager tools available from Python's project page, located at https://sourceforge.net/projects/python/.

The most important of the services now hosted at SourceForge is the Python CVS tree, the version-controlled repository containing the source code for Python. Previously, there were roughly 7 or so people who had write access to the CVS tree, and all patches had to be inspected and checked in by one of the people on this short list. Obviously, this wasn’t very scalable. By moving the CVS tree to SourceForge, it became possible to grant write access to more people; as of September 2000 there were 27 people able to check in changes, a fourfold increase. This makes possible large-scale changes that wouldn't be attempted if they'd have to be filtered through the small group of core developers. For example, one day Peter Schneider-Kamp took it into his head to drop K&R C compatibility and convert the C source for Python to ANSI C. After getting approval on the python-dev mailing list, he launched into a flurry of checkins that lasted about a week, other developers joined in to help, and the job was done. If there were only 5 people with write access, probably that task would have been viewed as “nice, but not worth the time and effort needed” and it would never have gotten done.

The shift to using SourceForge's services has resulted in a remarkable increase in the speed of development. Patches now get
submitted, commented on, revised by people other than the original submitter, and bounced back and forth between people until the patch is deemed worth checking in. Bugs are tracked in one central location and can be assigned to a specific person for fixing, and we can count the number of open bugs to measure progress. This didn’t come without a cost: developers now have more e-mail to deal with, more mailing lists to follow, and special tools had to be written for the new environment. For example, SourceForge sends default patch and bug notification e-mail messages that are completely unhelpful, so Ka-Ping Yee wrote an HTML screen-scraper that sends more useful messages.

The ease of adding code caused a few initial growing pains, such as code was checked in before it was ready or without getting clear agreement from the developer group. The approval process that has emerged is somewhat similar to that used by the Apache group. Developers can vote +1, +0, -0, or -1 on a patch; +1 and -1 denote acceptance or rejection, while +0 and -0 mean the developer is mostly indifferent to the change, though with a slight positive or negative slant. The most significant change from the Apache model is that the voting is essentially advisory, letting Guido van Rossum, who has Benevolent Dictator For Life status, know what the general opinion is. He can still ignore the result of a vote, and approve or reject a change even if the community disagrees with him.

Producing an actual patch is the last step in adding a new feature, and is usually easy compared to the earlier task of coming up with a good design. Discussions of new features can often explode into lengthy mailing list threads, making the discussion hard to follow, and no one can read every posting to python-dev. Therefore, a relatively formal process has been set up to write Python Enhancement Proposals (PEPs), modelled on the Internet RFC process. PEPs are draft documents that describe a proposed new feature, and are continually revised until the community reaches a consensus, either accepting or rejecting the proposal. Quoting from
the introduction to PEP 1, “PEP Purpose and Guidelines”:

PEP stands for Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python. The PEP should provide a concise technical specification of the feature and a rationale for the feature.

We intend PEPs to be the primary mechanisms for proposing new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

Read the rest of PEP 1 for the details of the PEP editorial process, style, and format. PEPs are kept in the Python CVS tree on SourceForge, though they're not part of the Python 2.0 distribution, and are also available in HTML form from https://www.python.org/dev/peps/. As of September 2000, there are 25 PEPS, ranging from PEP 201, “Lockstep Iteration”, to PEP 225, “Elementwise/Objectwise Operators”.


The largest new feature in Python 2.0 is a new fundamental data type: Unicode strings. Unicode uses 16-bit numbers to represent characters instead of the 8-bit number used by ASCII, meaning that 65,536 distinct characters can be supported.

The final interface for Unicode support was arrived at through countless often-stormy discussions on the python-dev mailing list, and mostly implemented by Marc-André Lemburg, based on a Unicode string type implementation by Fredrik Lundh. A detailed explanation of the interface was written up as PEP 100, “Python Unicode Integration”. This article will simply cover the most significant points about the Unicode interfaces.

In Python source code, Unicode strings are written as u"string". Arbitrary Unicode characters can be written using a new escape sequence, \uHHHH, where HHHH is a 4-digit hexadecimal number from 0000 to FFFF. The existing \xHHHH escape sequence can also be used, and octal escapes can be used for characters up to U+01FF, which is represented by \777.

Unicode strings, just like regular strings, are an immutable sequence type. They can be indexed and sliced, but not modified in place. Unicode strings have an encode([encoding]) method that returns an 8-bit string in the desired encoding. Encodings are named by strings, such as 'ascii', 'utf-8', 'iso-8859-1', or whatever. A codec API is defined for implementing and registering new encodings that are then available throughout a Python program. If an encoding isn’t specified, the default encoding is usually 7-bit ASCII, though it can be changed for your Python installation by calling the sys.setdefaultencoding(encoding)() function in a customised version of site.py.
Combining 8-bit and Unicode strings always coerces to Unicode, using the default ASCII encoding; the result of \('a' + u'bc'\) is \(u'abc'\).

New built-in functions have been added, and existing built-ins modified to support Unicode:

- **unichr(ch)** returns a Unicode string 1 character long, containing the character \(ch\).
- **ord(u)**, where \(u\) is a 1-character regular or Unicode string, returns the number of the character as an integer.
- **unicode(string [, encoding] [, errors] )** creates a Unicode string from an 8-bit string. **encoding** is a string naming the encoding to use. The **errors** parameter specifies the treatment of characters that are invalid for the current encoding; passing **'strict'** as the value causes an exception to be raised on any encoding error, while **'ignore'** causes errors to be silently ignored and **'replace'** uses U+FFFD, the official replacement character, in case of any problems.
- The **exec** statement, and various built-ins such as **eval()**, **getattr()**, and **setattr()** will also accept Unicode strings as well as regular strings. (It's possible that the process of fixing this missed some built-ins; if you find a built-in function that accepts strings but doesn't accept Unicode strings at all, please report it as a bug.)

A new module, **unicodedata**, provides an interface to Unicode character properties. For example, **unicodedata.category(u'A')** returns the 2-character string ‘Lu’, the ‘L’ denoting it’s a letter, and ‘u’ meaning that it’s uppercase. **unicodedata.bidirectional(u'\u0660')** returns ‘AN’, meaning that U+0660 is an Arabic number.

The **codecs** module contains functions to look up existing encodings
and register new ones. Unless you want to implement a new encoding, you'll most often use the `codecs.lookup(encoding)()` function, which returns a 4-element tuple: `(encode_func, decode_func, stream_reader, stream_writer)`.

- `encode_func` is a function that takes a Unicode string, and returns a 2-tuple `(string, length)`. `string` is an 8-bit string containing a portion (perhaps all) of the Unicode string converted into the given encoding, and `length` tells you how much of the Unicode string was converted.
- `decode_func` is the opposite of `encode_func`, taking an 8-bit string and returning a 2-tuple `(ustring, length)`, consisting of the resulting Unicode string `ustring` and the integer `length` telling how much of the 8-bit string was consumed.
- `stream_reader` is a class that supports decoding input from a stream. `stream_reader(file_obj)` returns an object that supports the `read()`, `readline()`, and `readlines()` methods. These methods will all translate from the given encoding and return Unicode strings.
- `stream_writer`, similarly, is a class that supports encoding output to a stream. `stream_writer(file_obj)` returns an object that supports the `write()` and `writelines()` methods. These methods expect Unicode strings, translating them to the given encoding on output.

For example, the following code writes a Unicode string into a file, encoding it as UTF-8:

```python
import codecs

unistr = u'\u0660\u2000ab ...'

(UTF8_encode, UTF8_decode, 
 UTF8_streamreader, UTF8_streamwriter) = codecs.lookup('UTF-8')

output = UTF8_streamwriter( open( '/tmp/output', 'wb' ) )
```
The following code would then read UTF-8 input from the file:

```python
output.write( unistr )
output.close()
```

```python
input = UTF8_streamreader( open( '/tmp/output', 'rb' ) )
print repr(input.read())
input.close()
```

Unicode-aware regular expressions are available through the `re` module, which has a new underlying implementation called SRE written by Fredrik Lundh of Secret Labs AB.

A `-u` command line option was added which causes the Python compiler to interpret all string literals as Unicode string literals. This is intended to be used in testing and future-proofing your Python code, since some future version of Python may drop support for 8-bit strings and provide only Unicode strings.
List Comprehensions

Lists are a workhorse data type in Python, and many programs manipulate a list at some point. Two common operations on lists are to loop over them, and either pick out the elements that meet a certain criterion, or apply some function to each element. For example, given a list of strings, you might want to pull out all the strings containing a given substring, or strip off trailing whitespace from each line.

The existing `map()` and `filter()` functions can be used for this purpose, but they require a function as one of their arguments. This is fine if there’s an existing built-in function that can be passed directly, but if there isn’t, you have to create a little function to do the required work, and Python’s scoping rules make the result ugly if the little function needs additional information. Take the first example in the previous paragraph, finding all the strings in the list containing a given substring. You could write the following to do it:

```
# Given the list L, make a list of all strings
# containing the substring S.
sublist = filter( lambda s, substring=S:
    string.find(s, substring) != -1,
    L)
```

Because of Python’s scoping rules, a default argument is used so that the anonymous function created by the `lambda` statement knows what substring is being searched for. List comprehensions make this cleaner:

```
sublist = [ s for s in L if string.find(s, S) != -1 ]
```

List comprehensions have the form:

```
[ expression for expr in sequence1
```
The `for...in` clauses contain the sequences to be iterated over. The sequences do not have to be the same length, because they are *not* iterated over in parallel, but from left to right; this is explained more clearly in the following paragraphs. The elements of the generated list will be the successive values of `expression`. The final `if` clause is optional; if present, `expression` is only evaluated and added to the result if `condition` is true.

To make the semantics very clear, a list comprehension is equivalent to the following Python code:

```python
for expr1 in sequence1:
    for expr2 in sequence2:
        ...
    for exprN in sequenceN:
        if (condition):
            # Append the value of
            # the expression to the
            # resulting list.
```

This means that when there are multiple `for...in` clauses, the resulting list will be equal to the product of the lengths of all the sequences. If you have two lists of length 3, the output list is 9 elements long:

```python
seq1 = 'abc'
seq2 = (1,2,3)
>>> [(x,y) for x in seq1 for y in seq2]
[('a', 1), ('a', 2), ('a', 3), ('b', 1), ('b', 2), ('b', 3), ('c', 2), ('c', 3)]
```

To avoid introducing an ambiguity into Python's grammar, if `expression` is creating a tuple, it must be surrounded with parentheses. The first list comprehension below is a syntax error,
while the second one is correct:

```python
# Syntax error
[ x, y for x in seq1 for y in seq2]
# Correct
[ (x, y) for x in seq1 for y in seq2]
```

The idea of list comprehensions originally comes from the functional programming language Haskell (https://www.haskell.org). Greg Ewing argued most effectively for adding them to Python and wrote the initial list comprehension patch, which was then discussed for a seemingly endless time on the python-dev mailing list and kept up-to-date by Skip Montanaro.
Augmented Assignment

Augmented assignment operators, another long-requested feature, have been added to Python 2.0. Augmented assignment operators include `+=`, `-=`, `*=` , and so forth. For example, the statement `a += 2` increments the value of the variable `a` by 2, equivalent to the slightly lengthier `a = a + 2`.

The full list of supported assignment operators is `+=`, `-=`, `*=` , `/=`, `%=`, `**=`, `&=`, `|=`, `^=`, `>>=`, and `<<=`. Python classes can override the augmented assignment operators by defining methods named `__iadd__()`, `__isub__()`, etc. For example, the following `Number` class stores a number and supports using `+=` to create a new instance with an incremented value.

```python
class Number:
    def __init__(self, value):
        self.value = value
    def __iadd__(self, increment):
        return Number( self.value + increment)

n = Number(5)
n += 3
print n.value
```

The `__iadd__()` special method is called with the value of the increment, and should return a new instance with an appropriately modified value; this return value is bound as the new value of the variable on the left-hand side.

Augmented assignment operators were first introduced in the C programming language, and most C-derived languages, such as `awk`, `C++`, `Java`, `Perl`, and `PHP` also support them. The augmented assignment patch was implemented by Thomas Wouters.
String Methods

Until now string-manipulation functionality was in the `string` module, which was usually a front-end for the `strop` module written in C. The addition of Unicode posed a difficulty for the `strop` module, because the functions would all need to be rewritten in order to accept either 8-bit or Unicode strings. For functions such as `string.replace()`, which takes 3 string arguments, that means eight possible permutations, and correspondingly complicated code.

Instead, Python 2.0 pushes the problem onto the string type, making string manipulation functionality available through methods on both 8-bit strings and Unicode strings.

```python
>>> 'andrew'.capitalize()
'Andrew'
>>> 'hostname'.replace('os', 'linux')
'hlinuxtname'
>>> 'moshe'.find('sh')
2
```

One thing that hasn't changed, a noteworthy April Fools' joke notwithstanding, is that Python strings are immutable. Thus, the string methods return new strings, and do not modify the string on which they operate.

The old `string` module is still around for backwards compatibility, but it mostly acts as a front-end to the new string methods.

Two methods which have no parallel in pre-2.0 versions, although they did exist in JPython for quite some time, are `startswith()` and `endswith()`. `s.startswith(t)` is equivalent to `s[:len(t)] == t`, while `s.endswith(t)` is equivalent to `s[-len(t):] == t`. 
One other method which deserves special mention is `join()`. The `join()` method of a string receives one parameter, a sequence of strings, and is equivalent to the `string.join()` function from the old `string` module, with the arguments reversed. In other words, `s.join(seq)` is equivalent to the old `string.join(seq, s)`. 
Garbage Collection of Cycles

The C implementation of Python uses reference counting to implement garbage collection. Every Python object maintains a count of the number of references pointing to itself, and adjusts the count as references are created or destroyed. Once the reference count reaches zero, the object is no longer accessible, since you need to have a reference to an object to access it, and if the count is zero, no references exist any longer.

Reference counting has some pleasant properties: it's easy to understand and implement, and the resulting implementation is portable, fairly fast, and reacts well with other libraries that implement their own memory handling schemes. The major problem with reference counting is that it sometimes doesn't realise that objects are no longer accessible, resulting in a memory leak. This happens when there are cycles of references.

Consider the simplest possible cycle, a class instance which has a reference to itself:

```python
instance = SomeClass()
instance.myself = instance
```

After the above two lines of code have been executed, the reference count of `instance` is 2; one reference is from the variable named 'instance', and the other is from the `myself` attribute of the instance.

If the next line of code is `del instance`, what happens? The reference count of `instance` is decreased by 1, so it has a reference count of 1; the reference in the `myself` attribute still exists. Yet the instance is no longer accessible through Python code, and it could be deleted. Several objects can participate in a cycle if they have
references to each other, causing all of the objects to be leaked.

Python 2.0 fixes this problem by periodically executing a cycle detection algorithm which looks for inaccessible cycles and deletes the objects involved. A new gc module provides functions to perform a garbage collection, obtain debugging statistics, and tuning the collector's parameters.

Running the cycle detection algorithm takes some time, and therefore will result in some additional overhead. It is hoped that after we've gotten experience with the cycle collection from using 2.0, Python 2.1 will be able to minimize the overhead with careful tuning. It's not yet obvious how much performance is lost, because benchmarking this is tricky and depends crucially on how often the program creates and destroys objects. The detection of cycles can be disabled when Python is compiled, if you can't afford even a tiny speed penalty or suspect that the cycle collection is buggy, by specifying the --without-cycle-gc switch when running the configure script.

Several people tackled this problem and contributed to a solution. An early implementation of the cycle detection approach was written by Toby Kelsey. The current algorithm was suggested by Eric Tiedemann during a visit to CNRI, and Guido van Rossum and Neil Schemenauer wrote two different implementations, which were later integrated by Neil. Lots of other people offered suggestions along the way; the March 2000 archives of the python-dev mailing list contain most of the relevant discussion, especially in the threads titled “Reference cycle collection for Python” and “Finalization again”.


Other Core Changes

Various minor changes have been made to Python’s syntax and built-in functions. None of the changes are very far-reaching, but they’re handy conveniences.

Minor Language Changes

A new syntax makes it more convenient to call a given function with a tuple of arguments and/or a dictionary of keyword arguments. In Python 1.5 and earlier, you’d use the `apply()` built-in function:

```python
apply(f, args, kw)
```
calls the function `f()` with the argument tuple `args` and the keyword arguments in the dictionary `kw`. `apply()` is the same in 2.0, but thanks to a patch from Greg Ewing, `f(*args, **kw)` is a shorter and clearer way to achieve the same effect. This syntax is symmetrical with the syntax for defining functions:

```python
def f(*args, **kw):
    # args is a tuple of positional args,
    # kw is a dictionary of keyword args
    ...
```

The `print` statement can now have its output directed to a file-like object by following the `print` with `>> file`, similar to the redirection operator in Unix shells. Previously you’d either have to use the `write()` method of the file-like object, which lacks the convenience and simplicity of `print`, or you could assign a new value to `sys.stdout` and then restore the old value. For sending output to standard error, it’s much easier to write this:

```python
print >> sys.stderr, "Warning: action field not supplied"
```

Modules can now be renamed on importing them, using the syntax
import module as name OR from module import name as othername.

The patch was submitted by Thomas Wouters.

A new format style is available when using the % operator; '%r' will insert the repr() of its argument. This was also added from symmetry considerations, this time for symmetry with the existing '%s' format style, which inserts the str() of its argument. For example, '%r %s' % ('abc', 'abc') returns a string containing 'abc'

abc.

Previously there was no way to implement a class that overrode Python’s built-in in operator and implemented a custom version. obj in seq returns true if obj is present in the sequence seq; Python computes this by simply trying every index of the sequence until either obj is found or an IndexError is encountered. Moshe Zadka contributed a patch which adds a __contains__() magic method for providing a custom implementation for in. Additionally, new built-in objects written in C can define what in means for them via a new slot in the sequence protocol.

Earlier versions of Python used a recursive algorithm for deleting objects. Deeply nested data structures could cause the interpreter to fill up the C stack and crash; Christian Tismer rewrote the deletion logic to fix this problem. On a related note, comparing recursive objects recursed infinitely and crashed; Jeremy Hylton rewrote the code to no longer crash, producing a useful result instead. For example, after this code:

```python
a = []
b = []
a.append(a)
b.append(b)
```

The comparison a==b returns true, because the two recursive data
structures are isomorphic. See the thread “trashcan and PR#7” in the April 2000 archives of the python-dev mailing list for the discussion leading up to this implementation, and some useful relevant links. Note that comparisons can now also raise exceptions. In earlier versions of Python, a comparison operation such as \texttt{cmp(a,b)} would always produce an answer, even if a user-defined \texttt{__cmp__()} method encountered an error, since the resulting exception would simply be silently swallowed.

Work has been done on porting Python to 64-bit Windows on the Itanium processor, mostly by Trent Mick of ActiveState. (Confusingly, \texttt{sys.platform} is still \texttt{'win32'} on Win64 because it seems that for ease of porting, MS Visual C++ treats code as 32 bit on Itanium.) PythonWin also supports Windows CE; see the Python CE page at \url{http://pythonce.sourceforge.net/} for more information.

Another new platform is Darwin/MacOS X; initial support for it is in Python 2.0. Dynamic loading works, if you specify “configure –with-dyld –with-suffix=.x”. Consult the README in the Python source distribution for more instructions.

An attempt has been made to alleviate one of Python’s warts, the often-confusing \texttt{NameError} exception when code refers to a local variable before the variable has been assigned a value. For example, the following code raises an exception on the \texttt{print} statement in both 1.5.2 and 2.0; in 1.5.2 a \texttt{NameError} exception is raised, while 2.0 raises a new \texttt{UnboundLocalError} exception. \texttt{UnboundLocalError} is a subclass of \texttt{NameError}, so any existing code that expects \texttt{NameError} to be raised should still work.

```python
def f():
    print "i=", i
    i = i + 1
f()
```
Two new exceptions, TabError and IndentationError, have been introduced. They're both subclasses of SyntaxError, and are raised when Python code is found to be improperly indented.

**Changes to Built-in Functions**

A new built-in, `zip(seq1, seq2, ...)()`, has been added. `zip()` returns a list of tuples where each tuple contains the i-th element from each of the argument sequences. The difference between `zip()` and `map(None, seq1, seq2)` is that `map()` pads the sequences with `None` if the sequences aren't all of the same length, while `zip()` truncates the returned list to the length of the shortest argument sequence.

The `int()` and `long()` functions now accept an optional “base” parameter when the first argument is a string. `int('123', 10)` returns 123, while `int('123', 16)` returns 291. `int(123, 16)` raises a TypeError exception with the message “can’t convert non-string with explicit base”.

A new variable holding more detailed version information has been added to the `sys` module. `sys.version_info` is a tuple `(major, minor, micro, level, serial)` For example, in a hypothetical 2.0.1beta1, `sys.version_info` would be `(2, 0, 1, 'beta', 1). level is a string such as "alpha", "beta", or "final" for a final release.

Dictionaries have an odd new method, `setdefault(key, default)()`, which behaves similarly to the existing `get()` method. However, if the key is missing, `setdefault()` both returns the value of `default` as `get()` would do, and also inserts it into the dictionary as the value for `key`. Thus, the following lines of code:

```python
if dict.has_key( key ): return dict[key]
```
else:
    dict[key] = []
return dict[key]

can be reduced to a single `return dict.setdefault(key, [])` statement.

The interpreter sets a maximum recursion depth in order to catch runaway recursion before filling the C stack and causing a core dump or GPF. Previously this limit was fixed when you compiled Python, but in 2.0 the maximum recursion depth can be read and modified using `sys.getrecursionlimit()` and `sys.setrecursionlimit()`. The default value is 1000, and a rough maximum value for a given platform can be found by running a new script, Misc/find_recursionlimit.py.
Porting to 2.0

New Python releases try hard to be compatible with previous releases, and the record has been pretty good. However, some changes are considered useful enough, usually because they fix initial design decisions that turned out to be actively mistaken, that breaking backward compatibility can’t always be avoided. This section lists the changes in Python 2.0 that may cause old Python code to break.

The change which will probably break the most code is tightening up the arguments accepted by some methods. Some methods would take multiple arguments and treat them as a tuple, particularly various list methods such as `append()` and `insert()`. In earlier versions of Python, if `L` is a list, `L.append( 1,2 )` appends the tuple `(1,2)` to the list. In Python 2.0 this causes a `TypeError` exception to be raised, with the message: ‘append requires exactly 1 argument; 2 given’. The fix is to simply add an extra set of parentheses to pass both values as a tuple: `L.append( (1,2) )`.

The earlier versions of these methods were more forgiving because they used an old function in Python’s C interface to parse their arguments; 2.0 modernizes them to use `PyArg_ParseTuple()`, the current argument parsing function, which provides more helpful error messages and treats multi-argument calls as errors. If you absolutely must use 2.0 but can’t fix your code, you can edit `Objects/listobject.c` and define the preprocessor symbol `NO_STRICT_LIST_APPEND` to preserve the old behaviour; this isn’t recommended.

Some of the functions in the `socket` module are still forgiving in this way. For example, `socket.connect( ('hostname', 25) )()` is the
correct form, passing a tuple representing an IP address, but
socket.connect( 'hostname', 25 )() also works.
socket.connect_ex() and socket.bind() are similarly easy-going.
2.0alpha1 tightened these functions up, but because the
documentation actually used the erroneous multiple argument form,
many people wrote code which would break with the stricter
checking. GvR backed out the changes in the face of public reaction,
so for the socket module, the documentation was fixed and the
multiple argument form is simply marked as deprecated; it will be
tightened up again in a future Python version.

The \x escape in string literals now takes exactly 2 hex digits.
Previously it would consume all the hex digits following the ‘x’ and
take the lowest 8 bits of the result, so \x123456 was equivalent to
\x56.

The AttributeError and NameError exceptions have a more friendly
error message, whose text will be something like 'Spam' instance
has no attribute 'eggs' or name 'eggs' is not defined. Previously
the error message was just the missing attribute name eggs, and
code written to take advantage of this fact will break in 2.0.

Some work has been done to make integers and long integers a bit
more interchangeable. In 1.5.2, large-file support was added for
Solaris, to allow reading files larger than 2 GiB; this made the tell()
method of file objects return a long integer instead of a regular
integer. Some code would subtract two file offsets and attempt to use
the result to multiply a sequence or slice a string, but this raised a
TypeError. In 2.0, long integers can be used to multiply or slice a
sequence, and it’ll behave as you’d intuitively expect it to; 3L * 'abc'
produces 'abcabcabc', and (0,1,2,3)[2L:4L] produces (2,3). Long
integers can also be used in various contexts where previously only
integers were accepted, such as in the seek() method of file objects,
and in the formats supported by the `%` operator (`%d`, `%i`, `%x`, etc.). For example, "%d" % 2L**64 will produce the string 18446744073709551616.

The subtlest long integer change of all is that the `str()` of a long integer no longer has a trailing ‘L’ character, though `repr()` still includes it. The ‘L’ annoyed many people who wanted to print long integers that looked just like regular integers, since they had to go out of their way to chop off the character. This is no longer a problem in 2.0, but code which does `str(longval)[:-1]` and assumes the ‘L’ is there, will now lose the final digit.

Taking the `repr()` of a float now uses a different formatting precision than `str()`. `repr()` uses `.17g` format string for C's `sprintf()`, while `str()` uses `.12g` as before. The effect is that `repr()` may occasionally show more decimal places than `str()`, for certain numbers. For example, the number 8.1 can’t be represented exactly in binary, so `repr(8.1)` is '8.0999999999999996', while `str(8.1)` is '8.1'.

The `-x` command-line option, which turned all standard exceptions into strings instead of classes, has been removed; the standard exceptions will now always be classes. The `exceptions` module containing the standard exceptions was translated from Python to a built-in C module, written by Barry Warsaw and Fredrik Lundh.
Extending/Embedding Changes

Some of the changes are under the covers, and will only be apparent to people writing C extension modules or embedding a Python interpreter in a larger application. If you aren’t dealing with Python’s C API, you can safely skip this section.

The version number of the Python C API was incremented, so C extensions compiled for 1.5.2 must be recompiled in order to work with 2.0. On Windows, it’s not possible for Python 2.0 to import a third party extension built for Python 1.5.x due to how Windows DLLs work, so Python will raise an exception and the import will fail.

Users of Jim Fulton’s ExtensionClass module will be pleased to find out that hooks have been added so that ExtensionClasses are now supported by `isinstance()` and `issubclass()`. This means you no longer have to remember to write code such as `if type(obj) == myExtensionClass`, but can use the more natural `if isinstance(obj, myExtensionClass)`.

The `Python/importdl.c` file, which was a mass of #ifdefs to support dynamic loading on many different platforms, was cleaned up and reorganised by Greg Stein. `importdl.c` is now quite small, and platform-specific code has been moved into a bunch of `Python/dynload_*.c` files. Another cleanup: there were also a number of `my*.h` files in the Include/ directory that held various portability hacks; they’ve been merged into a single file, `Include/pyport.h`.

Vladimir Marangozov’s long-awaited malloc restructuring was completed, to make it easy to have the Python interpreter use a custom allocator instead of C’s standard `malloc()`. For documentation, read the comments in `Include/pymem.h` and
Include/objimpl.h. For the lengthy discussions during which the interface was hammered out, see the Web archives of the 'patches' and 'python-dev' lists at python.org.

Recent versions of the GUSI development environment for MacOS support POSIX threads. Therefore, Python's POSIX threading support now works on the Macintosh. Threading support using the user-space GNU pth library was also contributed.

Threading support on Windows was enhanced, too. Windows supports thread locks that use kernel objects only in case of contention; in the common case when there's no contention, they use simpler functions which are an order of magnitude faster. A threaded version of Python 1.5.2 on NT is twice as slow as an unthreaded version; with the 2.0 changes, the difference is only 10%. These improvements were contributed by Yakov Markovitch.

Python 2.0's source now uses only ANSI C prototypes, so compiling Python now requires an ANSI C compiler, and can no longer be done using a compiler that only supports K&R C.

Previously the Python virtual machine used 16-bit numbers in its bytecode, limiting the size of source files. In particular, this affected the maximum size of literal lists and dictionaries in Python source; occasionally people who are generating Python code would run into this limit. A patch by Charles G. Waldman raises the limit from $2^{16}$ to $2^{32}$.

Three new convenience functions intended for adding constants to a module’s dictionary at module initialization time were added: `PyModule_AddObject()`, `PyModule_AddIntConstant()`, and `PyModule_AddStringConstant()`. Each of these functions takes a module object, a null-terminated C string containing the name to be added, and a third argument for the value to be assigned to the name. This third argument is, respectively, a Python object, a C long,
or a C string.

A wrapper API was added for Unix-style signal handlers. `PyOS_getsig()` gets a signal handler and `PyOS_setsig()` will set a new handler.
Distutils: Making Modules Easy to Install

Before Python 2.0, installing modules was a tedious affair – there was no way to figure out automatically where Python is installed, or what compiler options to use for extension modules. Software authors had to go through an arduous ritual of editing Makefiles and configuration files, which only really work on Unix and leave Windows and MacOS unsupported. Python users faced wildly differing installation instructions which varied between different extension packages, which made administering a Python installation something of a chore.

The SIG for distribution utilities, shepherded by Greg Ward, has created the Distutils, a system to make package installation much easier. They form the distutils package, a new part of Python's standard library. In the best case, installing a Python module from source will require the same steps: first you simply mean unpack the tarball or zip archive, and the run “python setup.py install”. The platform will be automatically detected, the compiler will be recognized, C extension modules will be compiled, and the distribution installed into the proper directory. Optional command-line arguments provide more control over the installation process, the distutils package offers many places to override defaults – separating the build from the install, building or installing in non-default directories, and more.

In order to use the Distutils, you need to write a setup.py script. For the simple case, when the software contains only .py files, a minimal setup.py can be just a few lines long:

```python
from distutils.core import setup
setup (name = "foo", version = "1.0",
       py_modules = ["module1", "module2"])
```
The setup.py file isn’t much more complicated if the software consists of a few packages:

```python
text
from distutils.core import setup
text
setup (name = "foo", version = "1.0",
        packages = ["package", "package.subpackage"])
```

A C extension can be the most complicated case; here’s an example taken from the PyXML package:

```python
text
from distutils.core import setup, Extension
text
expat_extension = Extension('xml.parsers.pyexpat',
                             define_macros = [("XML_NS", None)],
                             include_dirs = ['extensions/expat/xmltok',
                                            'extensions/expat/parse'],
                             sources = ['extensions/pyexpat.c',
                                        'extensions/expat/xmltok/xmltok.c',
                                        'extensions/expat/xmltok/xmlrole.c',],
                     )
text
setup (name = "PyXML", version = "0.5.4",
       ext_modules = [expat_extension])
```

The Distutils can also take care of creating source and binary distributions. The “sdist” command, run by “python setup.py sdist”, builds a source distribution such as foo-1.0.tar.gz. Adding new commands isn’t difficult, “bdist_rpm” and “bdist_wininst” commands have already been contributed to create an RPM distribution and a Windows installer for the software, respectively. Commands to create other distribution formats such as Debian packages and Solaris .pkg files are in various stages of development.

All this is documented in a new manual, *Distributing Python Modules*, that joins the basic set of Python documentation.
XML Modules

Python 1.5.2 included a simple XML parser in the form of the `xmlilib` module, contributed by Sjoerd Mullender. Since 1.5.2’s release, two different interfaces for processing XML have become common: SAX2 (version 2 of the Simple API for XML) provides an event-driven interface with some similarities to `xmlilib`, and the DOM (Document Object Model) provides a tree-based interface, transforming an XML document into a tree of nodes that can be traversed and modified. Python 2.0 includes a SAX2 interface and a stripped-down DOM interface as part of the `xml` package. Here we will give a brief overview of these new interfaces; consult the Python documentation or the source code for complete details. The Python XML SIG is also working on improved documentation.

SAX2 Support

SAX defines an event-driven interface for parsing XML. To use SAX, you must write a SAX handler class. Handler classes inherit from various classes provided by SAX, and override various methods that will then be called by the XML parser. For example, the `startElement()` and `endElement()` methods are called for every starting and end tag encountered by the parser, the `characters()` method is called for every chunk of character data, and so forth.

The advantage of the event-driven approach is that the whole document doesn’t have to be resident in memory at any one time, which matters if you are processing really huge documents. However, writing the SAX handler class can get very complicated if you’re trying to modify the document structure in some elaborate way.

For example, this little example program defines a handler that prints
a message for every starting and ending tag, and then parses the file hamlet.xml using it:

```python
from xml import sax

class SimpleHandler(sax.ContentHandler):
    def startElement(self, name, attrs):
        print 'Start of element:', name, attrs.keys()

    def endElement(self, name):
        print 'End of element:', name

# Create a parser object
parser = sax.make_parser()

# Tell it what handler to use
handler = SimpleHandler()
parser.setContentHandler( handler )

# Parse a file!
parser.parse( 'hamlet.xml' )
```

For more information, consult the Python documentation, or the XML HOWTO at http://pyxml.sourceforge.net/topics/howto/xml-howto.html.

**DOM Support**

The Document Object Model is a tree-based representation for an XML document. A top-level `Document` instance is the root of the tree, and has a single child which is the top-level `Element` instance. This `Element` has children nodes representing character data and any sub-elements, which may have further children of their own, and so forth. Using the DOM you can traverse the resulting tree any way you like, access element and attribute values, insert and delete nodes, and convert the tree back into XML.

The DOM is useful for modifying XML documents, because you can create a DOM tree, modify it by adding new nodes or rearranging
subtrees, and then produce a new XML document as output. You can also construct a DOM tree manually and convert it to XML, which can be a more flexible way of producing XML output than simply writing <tag1>...</tag1> to a file.

The DOM implementation included with Python lives in the xml.dom.minidom module. It's a lightweight implementation of the Level 1 DOM with support for XML namespaces. The parse() and parseString() convenience functions are provided for generating a DOM tree:

```python
from xml.dom import minidom
doc = minidom.parse('hamlet.xml')
```

doc is a Document instance. Document, like all the other DOM classes such as Element and Text, is a subclass of the Node base class. All the nodes in a DOM tree therefore support certain common methods, such as toxml() which returns a string containing the XML representation of the node and its children. Each class also has special methods of its own; for example, Element and Document instances have a method to find all child elements with a given tag name. Continuing from the previous 2-line example:

```python
perslist = doc.getElementsByTagName('PERSONA')
print perslist[0].toxml()
print perslist[1].toxml()
```

For the Hamlet XML file, the above few lines output:

```xml
<Persona>CLAUDIUS, king of Denmark. </Persona>
<Persona>HAMLET, son to the late, and nephew to the present king</Persona>
```

The root element of the document is available as doc.documentElement, and its children can be easily modified by deleting, adding, or removing nodes:
root = doc.documentElement

# Remove the first child
root.removeChild( root.childNodes[0] )

# Move the new first child to the end
root.appendChild( root.childNodes[0] )

# Insert the new first child (originally, # the third child) before the 20th child.
root.insertBefore( root.childNodes[0], root.childNodes[20] )

Again, I will refer you to the Python documentation for a complete listing of the different Node classes and their various methods.

**Relationship to PyXML**

The XML Special Interest Group has been working on XML-related Python code for a while. Its code distribution, called PyXML, is available from the SIG’s Web pages at [https://www.python.org/community/sigs/current/xml-sig](https://www.python.org/community/sigs/current/xml-sig). The PyXML distribution also used the package name `xml`. If you’ve written programs that used PyXML, you’re probably wondering about its compatibility with the 2.0 `xml` package.

The answer is that Python 2.0’s `xml` package isn’t compatible with PyXML, but can be made compatible by installing a recent version PyXML. Many applications can get by with the XML support that is included with Python 2.0, but more complicated applications will require that the full PyXML package will be installed. When installed, PyXML versions 0.6.0 or greater will replace the `xml` package shipped with Python, and will be a strict superset of the standard package, adding a bunch of additional features. Some of the additional features in PyXML include:

- 4DOM, a full DOM implementation from FourThought, Inc.
- The xmlproc validating parser, written by Lars Marius Garshol.
- The `sgm1op` parser accelerator module, written by Fredrik Lundh.
Lots of improvements and bugfixes were made to Python's extensive standard library; some of the affected modules include `readline`, `ConfigParser`, `cgi`, `calendar`, `posix`, `readline`, `xmllib`, `aifc`, `chunk`, `wave`, `random`, `shelve`, and `nntplib`. Consult the CVS logs for the exact patch-by-patch details.

Brian Gallew contributed OpenSSL support for the `socket` module. OpenSSL is an implementation of the Secure Socket Layer, which encrypts the data being sent over a socket. When compiling Python, you can edit `Modules/Setup` to include SSL support, which adds an additional function to the `socket` module: `socket.ssl(socket, keyfile, certfile)()`, which takes a socket object and returns an SSL socket. The `httplib` and `urllib` modules were also changed to support `https://` URLs, though no one has implemented FTP or SMTP over SSL.

The `httplib` module has been rewritten by Greg Stein to support HTTP/1.1. Backward compatibility with the 1.5 version of `httplib` is provided, though using HTTP/1.1 features such as pipelining will require rewriting code to use a different set of interfaces.

The `Tkinter` module now supports Tcl/Tk version 8.1, 8.2, or 8.3, and support for the older 7.x versions has been dropped. The Tkinter module now supports displaying Unicode strings in Tk widgets. Also, Fredrik Lundh contributed an optimization which makes operations like `create_line` and `create_polygon` much faster, especially when using lots of coordinates.

The `curses` module has been greatly extended, starting from Oliver Andrich's enhanced version, to provide many additional functions
from ncurses and SYSV curses, such as colour, alternative character set support, pads, and mouse support. This means the module is no longer compatible with operating systems that only have BSD curses, but there don’t seem to be any currently maintained OSes that fall into this category.

As mentioned in the earlier discussion of 2.0’s Unicode support, the underlying implementation of the regular expressions provided by the `re` module has been changed. SRE, a new regular expression engine written by Fredrik Lundh and partially funded by Hewlett Packard, supports matching against both 8-bit strings and Unicode strings.
New modules

A number of new modules were added. We'll simply list them with brief descriptions; consult the 2.0 documentation for the details of a particular module.

- **atexit**: For registering functions to be called before the Python interpreter exits. Code that currently sets `sys.exitfunc` directly should be changed to use the `atexit` module instead, importing `atexit` and calling `atexit.register()` with the function to be called on exit. (Contributed by Skip Montanaro.)
- **codecs, encodings, unicodedata**: Added as part of the new Unicode support.
- **filecmp**: Supersedes the old `cmp`, `cmpcache` and `dircmp` modules, which have now become deprecated. (Contributed by Gordon MacMillan and Moshe Zadka.)
- **gettext**: This module provides internationalization (I18N) and localization (L10N) support for Python programs by providing an interface to the GNU gettext message catalog library. (Integrated by Barry Warsaw, from separate contributions by Martin von Löwis, Peter Funk, and James Henstridge.)
- **linuxaudiodev**: Support for the `/dev/audio` device on Linux, a twin to the existing `sunaudiodev` module. (Contributed by Peter Bosch, with fixes by Jeremy Hylton.)
- **mmap**: An interface to memory-mapped files on both Windows and Unix. A file's contents can be mapped directly into memory, at which point it behaves like a mutable string, so its contents can be read and modified. They can even be passed to functions that expect ordinary strings, such as the `re` module. (Contributed by Sam Rushing, with some extensions by A.M. Kuchling.)
- **pyexpat**: An interface to the Expat XML parser. (Contributed by
• robotparser: Parse a robots.txt file, which is used for writing Web spiders that politely avoid certain areas of a Web site. The parser accepts the contents of a robots.txt file, builds a set of rules from it, and can then answer questions about the fetchability of a given URL. (Contributed by Skip Montanaro.)
• tabnanny: A module/script to check Python source code for ambiguous indentation. (Contributed by Tim Peters.)
• UserString: A base class useful for deriving objects that behave like strings.
• webbrowser: A module that provides a platform independent way to launch a web browser on a specific URL. For each platform, various browsers are tried in a specific order. The user can alter which browser is launched by setting the BROWSER environment variable. (Originally inspired by Eric S. Raymond's patch to urllib which added similar functionality, but the final module comes from code originally implemented by Fred Drake as Tools/idle/BrowserControl.py, and adapted for the standard library by Fred.)
• _winreg: An interface to the Windows registry. _winreg is an adaptation of functions that have been part of PythonWin since 1995, but has now been added to the core distribution, and enhanced to support Unicode. _winreg was written by Bill Tutt and Mark Hammond.
• zipfile: A module for reading and writing ZIP-format archives. These are archives produced by PKZIP on DOS/Windows or zip on Unix, not to be confused with gzip-format files (which are supported by the gzip module) (Contributed by James C. Ahlstrom.)
• imputil: A module that provides a simpler way for writing customised import hooks, in comparison to the existing ihooks module. (Implemented by Greg Stein, with much discussion on python-dev along the way.)
IDLE Improvements

IDLE is the official Python cross-platform IDE, written using Tkinter. Python 2.0 includes IDLE 0.6, which adds a number of new features and improvements. A partial list:

- UI improvements and optimizations, especially in the area of syntax highlighting and auto-indentation.
- The class browser now shows more information, such as the top level functions in a module.
- Tab width is now a user settable option. When opening an existing Python file, IDLE automatically detects the indentation conventions, and adapts.
- There is now support for calling browsers on various platforms, used to open the Python documentation in a browser.
- IDLE now has a command line, which is largely similar to the vanilla Python interpreter.
- Call tips were added in many places.
- IDLE can now be installed as a package.
- In the editor window, there is now a line/column bar at the bottom.
- Three new keystroke commands: Check module (Alt-F5), Import module (F5) and Run script (Ctrl-F5).
Deleted and Deprecated Modules

A few modules have been dropped because they're obsolete, or because there are now better ways to do the same thing. The stdwin module is gone; it was for a platform-independent windowing toolkit that's no longer developed.

A number of modules have been moved to the lib-old subdirectory: cmp, cmpcache, dircmp, dump, find, grep, packmail, poly, util, whatsound, zmod. If you have code which relies on a module that's been moved to lib-old, you can simply add that directory to sys.path to get them back, but you're encouraged to update any code that uses these modules.
Acknowledgements

The authors would like to thank the following people for offering suggestions on various drafts of this article: David Bolen, Mark Hammond, Gregg Hauser, Jeremy Hylton, Fredrik Lundh, Detlef Lannert, Aahz Maruch, Skip Montanaro, Vladimir Marangozov, Tobias Polzin, Guido van Rossum, Neil Schemenauer, and Russ Schmidt.
1. Whetting Your Appetite

If you do much work on computers, eventually you find that there’s some task you’d like to automate. For example, you may wish to perform a search-and-replace over a large number of text files, or rename and rearrange a bunch of photo files in a complicated way. Perhaps you’d like to write a small custom database, or a specialized GUI application, or a simple game.

If you’re a professional software developer, you may have to work with several C/C++/Java libraries but find the usual write/compile/test/re-compile cycle is too slow. Perhaps you’re writing a test suite for such a library and find writing the testing code a tedious task. Or maybe you’ve written a program that could use an extension language, and you don’t want to design and implement a whole new language for your application.

Python is just the language for you.

You could write a Unix shell script or Windows batch files for some of these tasks, but shell scripts are best at moving around files and changing text data, not well-suited for GUI applications or games. You could write a C/C++/Java program, but it can take a lot of development time to get even a first-draft program. Python is simpler to use, available on Windows, Mac OS X, and Unix operating systems, and will help you get the job done more quickly.

Python is simple to use, but it is a real programming language, offering much more structure and support for large programs than shell scripts or batch files can offer. On the other hand, Python also offers much more error checking than C, and, being a very-high-level language, it has high-level data types built in, such as flexible arrays and dictionaries. Because of its more general data types Python is applicable to a much larger problem domain than Awk or even Perl,
yet many things are at least as easy in Python as in those languages.

Python allows you to split your program into modules that can be reused in other Python programs. It comes with a large collection of standard modules that you can use as the basis of your programs — or as examples to start learning to program in Python. Some of these modules provide things like file I/O, system calls, sockets, and even interfaces to graphical user interface toolkits like Tk.

Python is an interpreted language, which can save you considerable time during program development because no compilation and linking is necessary. The interpreter can be used interactively, which makes it easy to experiment with features of the language, to write throw-away programs, or to test functions during bottom-up program development. It is also a handy desk calculator.

Python enables programs to be written compactly and readable. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons:

- the high-level data types allow you to express complex operations in a single statement;
- statement grouping is done by indentation instead of beginning and ending brackets;
- no variable or argument declarations are necessary.

Python is extensible: if you know how to program in C it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be available in binary form (such as a vendor-specific graphics library). Once you are really hooked, you can link the Python interpreter into an application written in C and use it as an extension or command language for that application.
By the way, the language is named after the BBC show “Monty Python’s Flying Circus” and has nothing to do with reptiles. Making references to Monty Python skits in documentation is not only allowed, it is encouraged!

Now that you are all excited about Python, you’ll want to examine it in some more detail. Since the best way to learn a language is to use it, the tutorial invites you to play with the Python interpreter as you read.

In the next chapter, the mechanics of using the interpreter are explained. This is rather mundane information, but essential for trying out the examples shown later.

The rest of the tutorial introduces various features of the Python language and system through examples, beginning with simple expressions, statements and data types, through functions and modules, and finally touching upon advanced concepts like exceptions and user-defined classes.
2. Using the Python Interpreter
2.1. Invoking the Interpreter

The Python interpreter is usually installed as `/usr/local/bin/python` on those machines where it is available; putting `/usr/local/bin` in your Unix shell's search path makes it possible to start it by typing the command

```
python
```

to the shell. Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., `/usr/local/python` is a popular alternative location.)

On Windows machines, the Python installation is usually placed in `C:\Python27`, though you can change this when you're running the installer. To add this directory to your path, you can type the following command into the command prompt in a DOS box:

```
set path=%path%;C:\python27
```

Typing an end-of-file character (Control-D on Unix, Control-Z on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn’t work, you can exit the interpreter by typing the following command: `quit()`.

The interpreter's line-editing features usually aren't very sophisticated. On Unix, whoever installed the interpreter may have enabled support for the GNU readline library, which adds more elaborate interactive editing and history features. Perhaps the quickest check to see whether command line editing is supported is typing Control-P to the first Python prompt you get. If it beeps, you have command line editing; see Appendix Interactive Input Editing
and History Substitution for an introduction to the keys. If nothing appears to happen, or if `^P` is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

The interpreter operates somewhat like the Unix shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a script from that file.

A second way of starting the interpreter is `python -c command [arg] ...`, which executes the statement(s) in `command`, analogous to the shell’s `-c` option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote `command` in its entirety with single quotes.

Some Python modules are also useful as scripts. These can be invoked using `python -m module [arg] ...`, which executes the source file for `module` as if you had spelled out its full name on the command line.

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing `-i` before the script.

All command-line options are described in Command line and environment.

### 2.1.1. Argument Passing

When known to the interpreter, the script name and additional arguments thereafter are turned into a list of strings and assigned to the `argv` variable in the `sys` module. You can access this list by
executing import sys. The length of the list is at least one; when no script and no arguments are given, sys.argv[0] is an empty string. When the script name is given as '-' (meaning standard input), sys.argv[0] is set to '-'. When -c command is used, sys.argv[0] is set to '-c'. When -m module is used, sys.argv[0] is set to the full name of the located module. Options found after -c command or -m module are not consumed by the Python interpreter’s option processing but left in sys.argv for the command or module to handle.

2.1.2. Interactive Mode

When commands are read from a tty, the interpreter is said to be in interactive mode. In this mode it prompts for the next command with the primary prompt, usually three greater-than signs (>>>); for continuation lines it prompts with the secondary prompt, by default three dots (...). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
python
Python 2.7 (#1, Feb 28 2010, 00:02:06)
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this if statement:

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Be careful not to fall off!"
... 
Be careful not to fall off!
```

For more on interactive mode, see Interactive Mode.
2.2. The Interpreter and Its Environment

2.2.1. Source Code Encoding

It is possible to use encodings different than ASCII in Python source files. The best way to do it is to put one more special comment line right after the `#!` line to define the source file encoding:

```
# -*- coding: encoding -*-
```

With that declaration, all characters in the source file will be treated as having the encoding `encoding`, and it will be possible to directly write Unicode string literals in the selected encoding. The list of possible encodings can be found in the Python Library Reference, in the section on `codecs`.

For example, to write Unicode literals including the Euro currency symbol, the ISO-8859-15 encoding can be used, with the Euro symbol having the ordinal value 164. This script, when saved in the ISO-8859-15 encoding, will print the value 8364 (the Unicode code point corresponding to the Euro symbol) and then exit:

```
# -*- coding: iso-8859-15 -*-
currency = u"€"
print ord(currency)
```

If your editor supports saving files as `UTF-8` with a UTF-8 `byte order mark` (aka BOM), you can use that instead of an encoding declaration. IDLE supports this capability if `Options/General/Default Source Encoding/UTF-8` is set. Notice that this signature is not understood in older Python releases (2.2 and earlier), and also not understood by the operating system for script files with `#!` lines (only
used on Unix systems).

By using UTF-8 (either through the signature or an encoding declaration), characters of most languages in the world can be used simultaneously in string literals and comments. Using non-ASCII characters in identifiers is not supported. To display all these characters properly, your editor must recognize that the file is UTF-8, and it must use a font that supports all the characters in the file.
3. An Informal Introduction to Python

In the following examples, input and output are distinguished by the presence or absence of prompts (`>>>` and `...`): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, `#`, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

```python
# this is the first comment
spam = 1  # and this is the second comment
    # ... and now a third!
text = "# This is not a comment because it's inside quotes."```
3.1. Using Python as a Calculator

Let’s try some simple Python commands. Start the interpreter and wait for the primary prompt, `>>>`. (It shouldn’t take long.)

3.1.1. Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` work just like in most other languages (for example, Pascal or C); parentheses `()` can be used for grouping. For example:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5.0*6) / 4
5.0
>>> 8 / 5.0
1.6
```

The integer numbers (e.g. `2`, `4`, `20`) have type `int`, the ones with a fractional part (e.g. `5.0`, `1.6`) have type `float`. We will see more about numeric types later in the tutorial.

The return type of a division `/` operation depends on its operands. If both operands are of type `int`, floor division is performed and an `int` is returned. If either operand is a `float`, classic division is performed and a `float` is returned. The `//` operator is also provided for doing floor division no matter what the operands are. The remainder can be calculated with the `%` operator:

```
>>> 17 / 3  # int / int -> int
5
```
With Python, it is possible to use the ** operator to calculate powers [1]:

```python
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```python
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

```python
>>> n  # try to access an undefined variable
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```python
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```
In interactive mode, the last printed expression is assigned to the variable \_\_. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```python
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

This variable should be treated as read-only by the user. Don’t explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to `int` and `float`, Python supports other types of numbers, such as `Decimal` and `Fraction`. Python also has built-in support for `complex numbers`, and uses the `j` or `J` suffix to indicate the imaginary part (e.g. `3+5j`).

### 3.1.2. Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes (\'\'\'\') or double quotes ("\"\") with the same result [2]. \ can be used to escape quotes:

```python
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \ to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
```
In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. While this might sometimes look different from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise it is enclosed in single quotes. The `print` statement produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```python
>>> "Isn't," she said.'
"Isn't," she said.
>>> print "Isn't," she said.'
"Isn't," she said.
>>> s = 'First line.
Second line.'  # \n means newline
>>> s  # without print, \n is included in the output
'First line..
Second line.'
>>> print s  # with print, \n produces a new line
First line.
Second line.
```

If you don't want characters prefaced by `\` to be interpreted as special characters, you can use raw strings by adding an `r` before the first quote:

```python
>>> print 'C:some\name'  # here \n means newline!
C:some\name
>>> print r'C:some\name'  # note the r before the quote
C:some\name
```

String literals can span multiple lines. One way is using triple-quotes: `"""...""""` or `'...''''. End of lines are automatically included in
the string, but it’s possible to prevent this by adding a \ at the end of
the line. The following example:

```python
print """
Usage: thingy [OPTIONS]
   -h                Display this usage message
   -H hostname       Hostname to connect to
"""

produces the following output (note that the initial newline is not
included):

```
Usage: thingy [OPTIONS]
   -h                Display this usage message
   -H hostname       Hostname to connect to
```

Strings can be concatenated (glued together) with the + operator,
and repeated with *:

```python
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Two or more string literals (i.e. the ones enclosed between quotes)
next to each other are automatically concatenated.

```python
>>> 'Py' 'thon'
'Python'
```

This only works with two literals though, not with variables or
expressions:

```python
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string
...
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
...
SyntaxError: invalid syntax
```
If you want to concatenate variables or a variable and a literal, use `+`:

```
>>> prefix + 'thon'
'Python'
```

This feature is particularly useful when you want to break long strings:

```
>>> text = ('Put several strings within parentheses ' +
          'to have them joined together. ')
>>> text
'Put several strings within parentheses to have them joined tog
```

Strings can be indexed (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

Indices may also be negative numbers, to start counting from the right:

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

Note that since -0 is the same as 0, negative indices start from -1.

In addition to indexing, **slicing** is also supported. While indexing is used to obtain individual characters, slicing allows you to obtain a
substring:

```python
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

Note how the start is always included, and the end always excluded. This makes sure that `s[:i] + s[i:]` is always equal to `s`:

```python
>>> word[:2] + word[2:]
'Python'
>>> word[4:] + word[4:]
'Python'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```python
>>> word[:2]   # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]   # characters from position 4 (included) to the end
'on'
>>> word[-2:]  # characters from the second-last (included) to the end
'on'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of `n` characters has index `n`, for example:

```
+------------------+------------------+
| P | y | t | h | o | n |
+------------------+------------------+
0    1    2    3    4    5    6
-6   -5   -4   -3   -2   -1
```

The first row of numbers gives the position of the indices 0...6 in the
string; the second row gives the corresponding negative indices. The slice from $i$ to $j$ consists of all characters between the edges labeled $i$ and $j$, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large will result in an error:

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

However, out of range slice indexes are handled gracefully when used for slicing:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python strings cannot be changed — they are immutable. Therefore, assigning to an indexed position in the string results in an error:

```
>>> word[0] = 'J'
...  
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
...  
TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```
The built-in function `len()` returns the length of a string:

```python
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

See also:

**Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange**

Strings, and the Unicode strings described in the next section, are examples of *sequence types*, and support the common operations supported by such types.

**String Methods**

Both strings and Unicode strings support a large number of methods for basic transformations and searching.

**Format String Syntax**

Information about string formatting with `str.format()`.

**String Formatting Operations**

The old formatting operations invoked when strings and Unicode strings are the left operand of the `%` operator are described in more detail here.

### 3.1.3. Unicode Strings

Starting with Python 2.0 a new data type for storing text data is available to the programmer: the Unicode object. It can be used to store and manipulate Unicode data (see [http://www.unicode.org](http://www.unicode.org)) and integrates well with the existing string objects, providing auto-conversions where necessary.

Unicode has the advantage of providing one ordinal for every character in every script used in modern and ancient texts.
Previously, there were only 256 possible ordinals for script characters. Texts were typically bound to a code page which mapped the ordinals to script characters. This lead to very much confusion especially with respect to internationalization (usually written as i18n — 'i' + 18 characters + 'n') of software. Unicode solves these problems by defining one code page for all scripts.

Creating Unicode strings in Python is just as simple as creating normal strings:

```
>>> u'Hello World !'
u'Hello World !'
```

The small 'u' in front of the quote indicates that a Unicode string is supposed to be created. If you want to include special characters in the string, you can do so by using the Python Unicode-Escape encoding. The following example shows how:

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

The escape sequence \u0020 indicates to insert the Unicode character with the ordinal value 0x0020 (the space character) at the given position.

Other characters are interpreted by using their respective ordinal values directly as Unicode ordinals. If you have literal strings in the standard Latin-1 encoding that is used in many Western countries, you will find it convenient that the lower 256 characters of Unicode are the same as the 256 characters of Latin-1.

For experts, there is also a raw mode just like the one for normal strings. You have to prefix the opening quote with ‘ur’ to have Python use the Raw-Unicode-Escape encoding. It will only apply the above \uxxxx conversion if there is an uneven number of backslashes in
The raw mode is most useful when you have to enter lots of backslashes, as can be necessary in regular expressions.

Apart from these standard encodings, Python provides a whole set of other ways of creating Unicode strings on the basis of a known encoding.

The built-in function `unicode()` provides access to all registered Unicode codecs (COders and DECodeRS). Some of the more well known encodings which these codecs can convert are Latin-1, ASCII, UTF-8, and UTF-16. The latter two are variable-length encodings that store each Unicode character in one or more bytes. The default encoding is normally set to ASCII, which passes through characters in the range 0 to 127 and rejects any other characters with an error. When a Unicode string is printed, written to a file, or converted with `str()`, conversion takes place using this default encoding.

To convert a Unicode string into an 8-bit string using a specific
encoding, Unicode objects provide an encode() method that takes
one argument, the name of the encoding. Lowercase names for
encodings are preferred.
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'

If you have data in a specific encoding and want to produce a
corresponding Unicode string from it, you can use the unicode()
function with the encoding name as the second argument.
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'

3.1.4. Lists
Python knows a number of compound data types, used to group
together other values. The most versatile is the list, which can be
written as a list of comma-separated values (items) between square
brackets. Lists might contain items of different types, but usually the
items all have the same type.
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]

Like strings (and all other built-in sequence type), lists can be
indexed and sliced:
>>>
1
>>>
25
>>>
[9,

squares[0]

# indexing returns the item

squares[-1]
squares[-3:]
16, 25]

# slicing returns a new list

All slice operations return a new list containing the requested


elements. This means that the following slice returns a new (shallow) copy of the list:

```python
>>> squares[:]
[1, 4, 9, 16, 25]
```

Lists also support operations like concatenation:

```python
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

```python
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the `append()` method (we will see more about methods later):

```python
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```python
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
```
The built-in function `len()` also applies to lists:

```python
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

It is possible to nest lists (create lists containing other lists), for example:

```python
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```
3.2. First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the *Fibonacci* series as follows:

```python
>>> # Fibonacci series:
>>> # the sum of two elements defines the next
>>> a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

This example introduces several new features.

- The first line contains a *multiple assignment*: the variables `a` and `b` simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.

- The *while* loop executes as long as the condition (here: `b < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: `<` (less than), `>` (greater than), `==`
(equal to), \(\leq\) (less than or equal to), \(\geq\) (greater than or equal to) and \(!=\) (not equal to).

- The body of the loop is **indented**: indentation is Python’s way of grouping statements. At the interactive prompt, you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; all decent text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.

- The **print** statement writes the value of the expression(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple expressions and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

```python
>>> i = 256*256
>>> print 'The value of i is', i
The value of i is 65536
```

A trailing comma avoids the newline after the output:

```python
>>> a, b = 0, 1
>>> while b < 1000:
...    print b,
...    a, b = b, a+b
...
```

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

Note that the interpreter inserts a newline before it prints the next prompt if the last line was not completed.

**Footnotes**
Since ** has higher precedence than -, -3**2 will be interpreted as -(3**2) and thus result in -9. To avoid this and get 9, you can use (-3)**2.

Unlike other languages, special characters such as \n have the same meaning with both single ('...') and double ("...") quotes. The only difference between the two is that within single quotes you don’t need to escape " (but you have to escape \') and vice versa.

[1] Since ** has higher precedence than -, -3**2 will be interpreted as -(3**2) and thus result in -9. To avoid this and get 9, you can use (-3)**2.

[2] Unlike other languages, special characters such as \n have the same meaning with both single ('...') and double ("...") quotes. The only difference between the two is that within single quotes you don’t need to escape " (but you have to escape \’) and vice versa.
4. More Control Flow Tools

Besides the `while` statement just introduced, Python knows the usual control flow statements known from other languages, with some twists.
4.1. if Statements

Perhaps the most well-known statement type is the if statement. For example:

```python
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...    x = 0
...    print 'Negative changed to zero'
... elif x == 0:
...    print 'Zero'
... elif x == 1:
...    print 'Single'
... else:
...    print 'More'
... More
```

There can be zero or more elif parts, and the else part is optional. The keyword ‘elif’ is short for ‘else if’, and is useful to avoid excessive indentation. An if ... elif ... elif ... sequence is a substitute for the switch or case statements found in other languages.
4.2. **for** Statements

The **for** statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's **for** statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```python
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print w, len(w)
... cat 3
   window 6
  defenestrate 12
```

If you need to modify the sequence you are iterating over while inside the loop (for example to duplicate selected items), it is recommended that you first make a copy. Iterating over a sequence does not implicitly make a copy. The slice notation makes this especially convenient:

```python
>>> for w in words[:]:  # Loop over a slice copy of the entire list.
...    if len(w) > 6:
...        words.insert(0, w)
...>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```
4.3. The `range()` Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates lists containing arithmetic progressions:

```python
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The given end point is never part of the generated list; `range(10)` generates a list of 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```python
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```python
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
... 0 Mary
 1 had
 2 a
 3 little
 4 lamb
```

In most such cases, however, it is convenient to use the `enumerate()` function, see Looping Techniques.
4.4. break and continue Statements, and else Clauses on Loops

The `break` statement, like in C, breaks out of the smallest enclosing `for` or `while` loop.

Loop statements may have an `else` clause; it is executed when the loop terminates through exhaustion of the list (with `for`) or when the condition becomes false (with `while`), but not when the loop is terminated by a `break` statement. This is exemplified by the following loop, which searches for prime numbers:

```python
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:
...         # loop fell through without finding a factor
...         print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the `else` clause belongs to the `for` loop, not the `if` statement.)

When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does that of `if` statements: a `try` statement’s `else` clause runs when no exception
occurs, and a loop’s else clause runs when no break occurs. For more on the try statement and exceptions, see Handling Exceptions.

The continue statement, also borrowed from C, continues with the next iteration of the loop:

```python
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print "Found an even number", num
...         continue
...     print "Found a number", num
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```
4.5. **pass** Statements

The **`pass`** statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```python
>>> while True:
    ...   pass  # Busy-wait for keyboard interrupt (Ctrl+C)
    ...
```

This is commonly used for creating minimal classes:

```python
>>> class MyEmptyClass:
    ...   pass
    ...
```

Another place **`pass`** can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The **`pass`** is silently ignored:

```python
>>> def initlog(*args):
    ...   pass  # Remember to implement this!
    ...
```
4.6. Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```python
>>> def fib(n):  # write Fibonacci series up to n
...     '''Print a Fibonacci series up to n.'''
...     a, b = 0, 1
...     while a < n:
...         print a,
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
...     fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function’s documentation string, or docstring. (More about docstrings can be found in the section Documentation Strings.) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it’s good practice to include docstrings in code that you write, so make a habit of it.

The execution of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function
(unless named in a \texttt{global} statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using \textit{call by value} (where the \textit{value} is always an object \textit{reference}, not the value of the object). [1] When a function calls another function, a new local symbol table is created for that call.

A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function. This serves as a general renaming mechanism:

\begin{verbatim}
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
\end{verbatim}

Coming from other languages, you might object that \texttt{fib} is not a function but a procedure since it doesn’t return a value. In fact, even functions without a \texttt{return} statement do return a value, albeit a rather boring one. This value is called \texttt{None} (it’s a built-in name). Writing the value \texttt{None} is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using \texttt{print}:

\begin{verbatim}
>>> fib(0)
>>> print fib(0)
None
\end{verbatim}

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:
This example, as usual, demonstrates some new Python features:

- The `return` statement returns with a value from a function. `return` without an expression argument returns `None`. Falling off the end of a function also returns `None`.
- The statement `result.append(a)` calls a method of the list object `result`. A method is a function that ‘belongs’ to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object’s type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using `classes`, see `Classes`) The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `result = result + [a]`, but more efficient.
4.7. More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

4.7.1. Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```python
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
    print complaint
```

This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

This example also introduces the `in` keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in
the *defining* scope, so that

```python
t = 5
def f(arg=i):
    print arg
i = 6
f()
```

will print 5.

**Important warning:** The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```python
def f(a, L=[]):
    L.append(a)
    return L
print f(1)
print f(2)
print f(3)
```

This will print

```
[1]
[1, 2]
[1, 2, 3]
```

If you don’t want the default to be shared between subsequent calls, you can write the function like this instead:

```python
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```
4.7.2. Keyword Arguments

Functions can also be called using keyword arguments of the form `kwarg=value`. For instance, the following function:

```python
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue '):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

```python
parrot(1000)  # 1 positional argument
parrot(voltage=1000)  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')  # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)  # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')  # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

but all the following calls would be invalid:

```python
parrot()  # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)  # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. `actor` is not a valid argument for the `parrot` function), and their order is not important. This also includes non-optional arguments (e.g. `parrot(voltage=1000) is valid too). No argument may receive a
value more than once. Here's an example that fails due to this restriction:

```python
>>> def function(a):
...   pass
... >>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see Mapping Types — dict) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (*name must occur before `**name`.) For example, if we define a function like this:

```python
def cheeseshop(kind, *arguments, **keywords):
  print "-- Do you have any", kind, "?"
  print "-- I'm sorry, we're all out of", kind
  for arg in arguments:
    print arg
  print "-" * 40
  keys = sorted(keywords.keys())
  for kw in keys:
    print kw, ":", keywords[kw]
```

It could be called like this:

```python
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

and of course it would print:
Note that the list of keyword argument names is created by sorting the result of the keywords dictionary's `keys()` method before printing its contents; if this is not done, the order in which the arguments are printed is undefined.

### 4.7.3. Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see Tuples and Sequences). Before the variable number of arguments, zero or more normal arguments may occur.

```python
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

### 4.7.4. Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in `range()` function expects separate `start` and `stop` arguments. If they are not available separately, write the function call with the `*`-operator to unpack the arguments out of a list or tuple:

```python
>>> range(3, 6)  # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]`
In the same fashion, dictionaries can deliver keyword arguments with the **-operator:

```python
>>> d = {'voltage': 'four million', 'state': 'bleedin' demised'}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised!
```

4.7.5. Lambda Expressions

Small anonymous functions can be created with the `lambda` keyword. This function returns the sum of its two arguments: `lambda a, b: a+b`. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope:

```python
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

The above example uses a lambda expression to return a function. Another use is to pass a small function as an argument:
4.7.6. Documentation Strings

There are emerging conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object’s purpose. For brevity, it should not explicitly state the object’s name or type, since these are available by other means (except if the name happens to be a verb describing a function’s operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line after the first line of the string determines the amount of indentation for the entire documentation string. (We can’t use the first line since it is generally adjacent to the string’s opening quotes so its indentation is not apparent in the string literal.) Whitespace “equivalent” to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Here is an example of a multi-line docstring:

```python
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```
```python
>>> def my_function():
...     """Do nothing, but document it.
...     """
...     No, really, it doesn't do anything.
...     ""
...     pass
...
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesn't do anything.
```
Now that you are about to write longer, more complex pieces of Python, it is a good time to talk about coding style. Most languages can be written (or more concise, formatted) in different styles; some are more readable than others. Making it easy for others to read your code is always a good idea, and adopting a nice coding style helps tremendously for that.

For Python, PEP 8 has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point; here are the most important points extracted for you:

- Use 4-space indentation, and no tabs.

  4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.

- Wrap lines so that they don’t exceed 79 characters.

  This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.

- Use blank lines to separate functions and classes, and larger blocks of code inside functions.

- When possible, put comments on a line of their own.

- Use docstrings.

- Use spaces around operators and after commas, but not directly inside bracketing constructs: \( a = f(1, 2) + g(3, 4) \).
Name your classes and functions consistently; the convention is to use camelCase for classes and lower_case_with_underscores for functions and methods. Always use self as the name for the first method argument (see A First Look at Classes for more on classes and methods).

Don’t use fancy encodings if your code is meant to be used in international environments. Plain ASCII works best in any case.

Footnotes

Actually, call by object reference would be a better description, since if a mutable object is passed, the caller will see any changes the callee makes to it (items inserted into a list).
5. Data Structures

This chapter describes some things you’ve learned about already in more detail, and adds some new things as well.
5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

\( \text{list.} \) \text{append}(x) \\
Add an item to the end of the list; equivalent to \( a[\text{len}(a):] = [x] \).

\( \text{list.} \) \text{extend}(L) \\
Extend the list by appending all the items in the given list; equivalent to \( a[\text{len}(a):] = L \).

\( \text{list.} \) \text{insert}(i, x) \\
Insert an item at a given position. The first argument is the index of the element before which to insert, so \( a.\text{insert}(0, \ x) \) inserts at the front of the list, and \( a.\text{insert}(\text{len}(a), \ x) \) is equivalent to \( a.\text{append}(x) \).

\( \text{list.} \) \text{remove}(x) \\
Remove the first item from the list whose value is \( x \). It is an error if there is no such item.

\( \text{list.} \) \text{pop}([i]) \\
Remove the item at the given position in the list, and return it. If no index is specified, \( a.\text{pop()} \) removes and returns the last item in the list. (The square brackets around the \( i \) in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

\( \text{list.} \) \text{index}(x) \\
Return the index in the list of the first item whose value is \( x \). It is an error if there is no such item.
list. **count**(\(x\))

Return the number of times \(x\) appears in the list.

list. **sort**(\(cmp=None, \ key=None, \ reverse=False\))

Sort the items of the list in place (the arguments can be used for sort customization, see **sorted()** for their explanation).

list. **reverse**()

Reverse the elements of the list, in place.

An example that uses most of the list methods:

```python
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]
```

You might have noticed that methods like `insert`, `remove` or `sort` that only modify the list have no return value printed – they return the default `None`. [1] This is a design principle for all mutable data structures in Python.
5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```python
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2. Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends. For example:

```python
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")  # Terry arrives
>>> queue.append("Graham")  # Graham arrives
```
5.1.3. Functional Programming Tools

There are three built-in functions that are very useful when used with lists: filter(), map(), and reduce().

filter(function, sequence) returns a sequence consisting of those items from the sequence for which function(item) is true. If sequence is a str, unicode or tuple, the result will be of the same type; otherwise, it is always a list. For example, to compute a sequence of numbers divisible by 3 or 5:

```python
>>> def f(x): return x % 3 == 0 or x % 5 == 0
... >>> filter(f, range(2, 25))
[3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24]
```

map(function, sequence) calls function(item) for each of the sequence's items and returns a list of the return values. For example, to compute some cubes:

```python
>>> def cube(x): return x*x*x
... >>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

More than one sequence may be passed; the function must then have as many arguments as there are sequences and is called with the corresponding item from each sequence (or None if some sequence is shorter than another). For example:
reduce(function, sequence) returns a single value constructed by calling the binary function function on the first two items of the sequence, then on the result and the next item, and so on. For example, to compute the sum of the numbers 1 through 10:

```python
>>> def add(x, y): return x+y
... >>> reduce(add, range(1, 11))
55
```

If there's only one item in the sequence, its value is returned; if the sequence is empty, an exception is raised.

A third argument can be passed to indicate the starting value. In this case the starting value is returned for an empty sequence, and the function is first applied to the starting value and the first sequence item, then to the result and the next item, and so on. For example,

```python
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
... >>> sum(range(1, 11))
55
>>> sum([])
0
```

Don’t use this example’s definition of sum(): since summing numbers is such a common need, a built-in function sum(sequence) is already provided, and works exactly like this.

## 5.1.4. List Comprehensions
List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```python
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
... >>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can obtain the same result with:

```python
squares = [x**2 for x in range(10)]
```

This is also equivalent to `squares = map(lambda x: x**2, range(10))`, but it's more concise and readable.

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal:

```python
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

and it's equivalent to:

```python
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
```
Note how the order of the `for` and `if` statements is the same in both these snippets.

If the expression is a tuple (e.g. the `(x, y)` in the previous example), it must be parenthesized.

```python
vec = [-4, -2, 0, 2, 4]
# create a new list with the values doubled
[x*2 for x in vec]
[-8, -4, 0, 4, 8]
# filter the list to exclude negative numbers
[x for x in vec if x >= 0]
[0, 2, 4]
# apply a function to all the elements
[abs(x) for x in vec]
[4, 2, 0, 2, 4]
# call a method on each element
freshfruit = ['banana', 'loganberry', 'passion fruit']
[weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
# create a list of 2-tuples like (number, square)
[(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
# the tuple must be parenthesized, otherwise an error is raised
[x, x**2 for x in range(6)]
File "<stdin>", line 1
  [x, x**2 for x in range(6)]
^  SyntaxError: invalid syntax
# flatten a list using a listcomp with two 'for'
vec = [[1,2,3], [4,5,6], [7,8,9]]
[num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehensions can contain complex expressions and nested functions:

```python
from math import pi
```
5.1.4.1. Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Consider the following example of a 3x4 matrix implemented as a list of 3 lists of length 4:

```python
>>> matrix = [
... [1, 2, 3, 4],
... [5, 6, 7, 8],
... [9, 10, 11, 12],
...]
```

The following list comprehension will transpose rows and columns:

```python
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

As we saw in the previous section, the nested listcomp is evaluated in the context of the `for` that follows it, so this example is equivalent to:

```python
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...     transposed_row = []
...     for row in matrix:
...         # the following 3 lines implement the nested listcomp
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...     transposed_row = []
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

which, in turn, is the same as:

```python
>>> transposed = []
>>> for i in range(4):
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```
\begin{verbatim}
... transposed_row.append(row[i])
... transposed.append(transposed_row)
...
```python
In the real world, you should prefer built-in functions to complex flow statements. The \texttt{zip()} function would do a great job for this use case:

```python
>>> zip(*matrix)
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]

See Unpacking Argument Lists for details on the asterisk in this line.
```
5.2. The `del` statement

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```python
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` can also be used to delete entire variables:

```python
>>> del a
```

Referencing the name `a` hereafter is an error (at least until another value is assigned to it). We'll find other uses for `del` later.
5.3. Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of sequence data types (see Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange). Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the tuple.

A tuple consists of a number of values separated by commas, for instance:

```python
>>> t = 12345, 54321, 'hello'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([[1, 2, 3], [3, 2, 1]])
```

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.
Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking (see later in this section) or indexing (or even by attribute in the case of namedtuples). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

```python
>>> empty = ()
>>> singleton = 'hello',  # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

The statement `t = 12345, 54321, 'hello!'` is an example of tuple packing: the values `12345, 54321` and `'hello!'` are packed together in a tuple. The reverse operation is also possible:

```python
>>> x, y, z = t
```

This is called, appropriately enough, sequence unpacking and works for any sequence on the right-hand side. Sequence unpacking requires the list of variables on the left to have the same number of elements as the length of the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.
5.4. Sets

Python also includes a data type for sets. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the \texttt{set()} function can be used to create sets. Note: to create an empty set you have to use \texttt{set()}, not \texttt{{}}; the latter creates an empty dictionary, a data structure that we discuss in the next section.

Here is a brief demonstration:

```python
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)  # create a set without duplicates
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit  # fast membership testing
True
>>> 'crabgrass' in fruit
False

>>> # Demonstrate set operations on unique letters from two words...
...  
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a  # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b  # letters in a but not in b
set(['a', 'd'])
>>> a | b  # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b  # letters in both a and b
set(['a', 'c'])
>>> a ^ b  # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```
Similarly to list comprehensions, set comprehensions are also supported:

```python
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
set(['r', 'd'])
```
5.5. Dictionaries

Another useful data type built into Python is the *dictionary* (see Mapping Types — dict). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

It is best to think of a dictionary as an unordered set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

The `keys()` method of a dictionary object returns a list of all the keys used in the dictionary, in arbitrary order (if you want it sorted, just apply the `sorted()` function to it). To check whether a single key is in the dictionary, use the `in` keyword.

Here is a small example using a dictionary:
The `dict()` constructor builds dictionaries directly from sequences of key-value pairs:

```python
dict([(\'sape\', 4139), (\'guido\', 4127), (\'jack\', 4098)])
{'sape': 4139, \'jack\': 4098, \'guido\': 4127}
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```python
{x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```python
dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, \'jack\': 4098, \'guido\': 4127}
```
5.6. Looping Techniques

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function.

```python
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
    ...
    print i, v
...
0 tic
1 tac
2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the `zip()` function.

```python
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
    ...
    print 'What is your {0}?  It is {1}.' .format(q, a)
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the `reversed()` function.

```python
>>> for i in reversed(xrange(1,10,2)):
    ...
    print i
...
9
7
5
3
1
```

To loop over a sequence in sorted order, use the `sorted()` function.
which returns a new sorted list while leaving the source unaltered.

```python
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...    print f
...apple
    banana
    orange
    pear
```

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the `iteritems()` method.

```python
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...    print k, v
...gallahad the pure
    robin the brave
```

It is sometimes tempting to change a list while you are looping over it; however, it is often simpler and safer to create a new list instead.

```python
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN')]
>>> filtered_data = []
>>> for value in raw_data:
...    if not math.isnan(value):
...        filtered_data.append(value)
...>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```
5.7. More on Conditions

The conditions used in `while` and `if` statements can contain any operators, not just comparisons.

The comparison operators `in` and `not in` check whether a value occurs (does not occur) in a sequence. The operators `is` and `is not` compare whether two objects are really the same object; this only matters for mutable objects like lists. All comparison operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained. For example, `a < b == c` tests whether `a` is less than `b` and moreover `b` equals `c`.

Comparisons may be combined using the Boolean operators `and` and `or`, and the outcome of a comparison (or of any other Boolean expression) may be negated with `not`. These have lower priorities than comparison operators; between them, `not` has the highest priority and `or` the lowest, so that `A and not B or C` is equivalent to `(A and (not B)) or C`. As always, parentheses can be used to express the desired composition.

The Boolean operators `and` and `or` are so-called *short-circuit* operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if `A` and `C` are true but `B` is false, `A and B and C` does not evaluate the expression `C`. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example,
Note that in Python, unlike C, assignment cannot occur inside expressions. C programmers may grumble about this, but it avoids a common class of problems encountered in C programs: typing `=` in an expression when `==` was intended.
5.8. Comparing Sequences and Other Types

Sequence objects may be compared to other objects with the same sequence type. The comparison uses lexicographical ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the ASCII ordering for individual characters. Some examples of comparisons between sequences of the same type:

\[
\begin{align*}
(1, 2, 3) &< (1, 2, 4) \\
[1, 2, 3] &< [1, 2, 4] \\
'ABC' &< 'C' < 'Pascal' < 'Python' \\
(1, 2, 3, 4) &< (1, 2, 4) \\
(1, 2) &< (1, 2, -1) \\
(1, 2, 3) &= (1.0, 2.0, 3.0) \\
(1, 2, ('aa', 'ab')) &< (1, 2, ('abc', 'a'), 4)
\end{align*}
\]

Note that comparing objects of different types is legal. The outcome is deterministic but arbitrary: the types are ordered by their name. Thus, a list is always smaller than a string, a string is always smaller than a tuple, etc. [1] Mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc.

Footnotes

The rules for comparing objects of different types should not be
relied upon; they may change in a future version of the language.
6. Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a script. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you’ve written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; definitions from a module can be imported into other modules or into the main module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module’s name (as a string) is available as the value of the global variable __name__. For instance, use your favorite text editor to create a file called fibo.py in the current directory with the following contents:

```python
# Fibonacci numbers module

def fib(n):  # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n):  # return Fibonacci series up to n
    result = []
```
Now enter the Python interpreter and import this module with the following command:

```python
>>> import fibo
```

This does not enter the names of the functions defined in `fibo` directly in the current symbol table; it only enters the module name `fibo` there. Using the module name you can access the functions:

```python
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```python
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```
6.1. More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement. [1] (They are also run if the file is executed as a script.)

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module’s global variables with the same notation used to refer to its functions, modname.itemname.

Modules can import other modules. It is customary but not required to place all import statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module’s global symbol table.

There is a variant of the import statement that imports names from a module directly into the importing module’s symbol table. For example:

```python
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, fibo is not defined).

There is even a variant to import all names that a module defines:
This imports all names except those beginning with an underscore (_).

Note that in general the practice of importing * from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

**Note:** For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it’s just one module you want to test interactively, use `reload()`, e.g. `reload(modulename)`.

### 6.1.1. Executing modules as scripts

When you run a Python module with

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:
If the module is imported, the code is not run:

```python
>>> import fibo
``` 

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

### 6.1.2. The Module Search Path

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- the directory containing the input script (or the current directory).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- the installation-dependent default.

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See section [Standard Modules](#) for more information.

### 6.1.3. “Compiled” Python files

As an important speed-up of the start-up time for short programs that
use a lot of standard modules, if a file called `spam.pyc` exists in the
directory where `spam.py` is found, this is assumed to contain an
already-“byte-compiled” version of the module `spam`. The
modification time of the version of `spam.py` used to create `spam.pyc` is
recorded in `spam.pyc`, and the `.pyc` file is ignored if these don’t
match.

Normally, you don’t need to do anything to create the `spam.pyc` file.
Whenever `spam.py` is successfully compiled, an attempt is made to
write the compiled version to `spam.pyc`. It is not an error if this
attempt fails; if for any reason the file is not written completely, the
resulting `spam.pyc` file will be recognized as invalid and thus ignored
later. The contents of the `spam.pyc` file are platform independent, so
a Python module directory can be shared by machines of different
architectures.

Some tips for experts:

- When the Python interpreter is invoked with the `-o` flag, 
  optimized code is generated and stored in `.pyo` files. The
  optimizer currently doesn’t help much; it only removes `assert`
  statements. When `-o` is used, all bytecode is optimized; `.pyc`
  files are ignored and `.py` files are compiled to optimized
  bytecode.

- Passing two `-o` flags to the Python interpreter (`-oo`) will cause
  the bytecode compiler to perform optimizations that could in
  some rare cases result in malfunctioning programs. Currently
  only `__doc__` strings are removed from the bytecode, resulting in
  more compact `.pyo` files. Since some programs may rely on
  having these available, you should only use this option if you
  know what you’re doing.
- A program doesn't run any faster when it is read from a `.pyc` or `.pyo` file than when it is read from a `.py` file; the only thing that's faster about `.pyc` or `.pyo` files is the speed with which they are loaded.

- When a script is run by giving its name on the command line, the bytecode for the script is never written to a `.pyc` or `.pyo` file. Thus, the startup time of a script may be reduced by moving most of its code to a module and having a small bootstrap script that imports that module. It is also possible to name a `.pyc` or `.pyo` file directly on the command line.

- It is possible to have a file called `spam.pyc` (or `spam.pyo` when `-O` is used) without a file `spam.py` for the same module. This can be used to distribute a library of Python code in a form that is moderately hard to reverse engineer.

- The module `compileall` can create `.pyc` files (or `.pyo` files when `-O` is used) for all modules in a directory.
6.2. Standard Modules

Python comes with a library of standard modules, described in a separate document, the Python Library Reference (“Library Reference” hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the \texttt{winreg} module is only provided on Windows systems. One particular module deserves some attention: \texttt{sys}, which is built into every Python interpreter. The variables \texttt{sys.ps1} and \texttt{sys.ps2} define the strings used as primary and secondary prompts:

```python
>>> import sys
>>> sys.ps1
'>>>'
>>> sys.ps2
'...
>>> sys.ps1 = 'C>'
C> print 'Yuck!'
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

The variable \texttt{sys.path} is a list of strings that determines the interpreter’s search path for modules. It is initialized to a default path taken from the environment variable \texttt{PYTHONPATH}, or from a built-in default if \texttt{PYTHONPATH} is not set. You can modify it using standard list operations:

```python
>>> import sys
```
>>> sys.path.append('/ufs/guido/lib/python')
6.3. The `dir()` Function

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```python
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
```

Without arguments, `dir()` lists the names you have defined currently:

```python
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', '__package__', 'a', 'fib', 'fibo',
```

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. If
you want a list of those, they are defined in the standard module
__builtin__:

```python
>>> import __builtin__
>>> dir(__builtin__)
```
6.4. Packages

Packages are a way of structuring Python’s module namespace by using “dotted module names”. For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other’s global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other’s module names.

Suppose you want to design a collection of modules (a “package”) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here’s a possible structure for your package (expressed in terms of a hierarchical filesystem):

```
sound/
  __init__.py
  formats/
    __init__.py
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
  effects/
```

Top-level package
Initialize the sound package
Subpackage for file format conversion
Subpackage for sound effects
When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```python
import sound.effects.echo
```

This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

```python
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

An alternative way of importing the submodule is:

```python
from sound.effects import echo
```
This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

```python
echo.echofilter(input, output, delay=0.7, atten=4)
```

Yet another variation is to import the desired function or variable directly:

```python
from sound.effects.echo import echofilter
```

Again, this loads the submodule `echo`, but this makes its function `echofilter()` directly available:

```python
echofilter(input, output, delay=0.7, atten=4)
```

Note that when using `from package import item`, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an `ImportError` exception is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but can’t be a class or function or variable defined in the previous item.

### 6.4.1. Importing * From a Package

Now what happens when the user writes `from sound.effects import *`? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing submodules might have unwanted side-effects that should only happen
when the sub-module is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The `import` statement uses the following convention: if a package's `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when `from package import *` is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing * from their package. For example, the file `sound/effects/__init__.py` could contain the following code:

```
__all__ = ["echo", "surround", "reverse"]
```

This would mean that `from sound.effects import *` would import the three named submodules of the `sound` package.

If `__all__` is not defined, the statement `from sound.effects import *` does not import all submodules from the package `sound.effects` into the current namespace; it only ensures that the package `sound.effects` has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by `__init__.py`. It also includes any submodules of the package that were explicitly loaded by previous `import` statements. Consider this code:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In this example, the `echo` and `surround` modules are imported in the current namespace because they are defined in the `sound.effects`
package when the `from...import` statement is executed. (This also works when `__all__` is defined.)

Although certain modules are designed to export only names that follow certain patterns when you use `import *`, it is still considered bad practice in production code.

Remember, there is nothing wrong with using `from Package import specific_submodule`! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

### 6.4.2. Intra-package References

The submodules often need to refer to each other. For example, the `surround` module might use the `echo` module. In fact, such references are so common that the `import` statement first looks in the containing package before looking in the standard module search path. Thus, the `surround` module can simply use `import echo` or `from echo import echofilter`. If the imported module is not found in the current package (the package of which the current module is a submodule), the `import` statement looks for a top-level module with the given name.

When packages are structured into subpackages (as with the `sound` package in the example), you can use absolute imports to refer to submodules of siblings packages. For example, if the module `sound.filters.vocoder` needs to use the `echo` module in the `sound.effects` package, it can use `from sound.effects import echo`.

Starting with Python 2.5, in addition to the implicit relative imports described above, you can write explicit relative imports with the `from module import name` form of import statement. These explicit relative
imports use leading dots to indicate the current and parent packages involved in the relative import. From the surround module for example, you might use:

```python
from . import echo
from .. import formats
from ..filters import equalizer
```

Note that both explicit and implicit relative imports are based on the name of the current module. Since the name of the main module is always "__main__", modules intended for use as the main module of a Python application should always use absolute imports.

### 6.4.3. Packages in Multiple Directories

Packages support one more special attribute, __path__. This is initialized to be a list containing the name of the directory holding the package’s __init__.py before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

While this feature is not often needed, it can be used to extend the set of modules found in a package.

**Footnotes**

[1] In fact function definitions are also ‘statements’ that are ‘executed’; the execution of a module-level function definition enters the function name in the module’s global symbol table.
7. Input and Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.
### 7.1. Fancier Output Formatting

So far we’ve encountered two ways of writing values: *expression statements* and the `print` statement. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you’ll want more control over the formatting of your output than simply printing space-separated values. There are two ways to format your output; the first way is to do all the string handling yourself; using string slicing and concatenation operations you can create any layout you can imagine. The string types have some methods that perform useful operations for padding strings to a given column width; these will be discussed shortly. The second way is to use the `str.format()` method.

The `string` module contains a `Template` class which offers yet another way to substitute values into strings.

One question remains, of course: how do you convert values to strings? Luckily, Python has ways to convert any value to a string: pass it to the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don’t have a particular representation for human consumption, `str()` will return the same value as `repr()`. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings and floating point numbers, in
particular, have two distinct representations.

Some examples:

```python
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1.0/7.0)
'0.142857142857'
>>> repr(1.0/7.0)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y)
>>> print s
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print hellos
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

Here are two ways to write a table of squares and cubes:

```python
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # Note trailing comma on previous line
...     print repr(x*x*x).rjust(4)
...
1   1   1
2   4   8
3   9  27
4  16  64
5  25 125
6  36 216
7  49 343
8  64 512
9  81 729
10 100 1000
```
for x in range(1, 11):
...   print '{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x)
...
1 1  1
2 4  8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000

(Note that in the first example, one space between each column was added by the way print works: it always adds spaces between its arguments.)

This example demonstrates the str.rjust() method of string objects, which right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods str.ljust() and str.center(). These methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column lay-out but that's usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in x.ljust(n)[:n].)

There is another method, str.zfill(), which pads a numeric string on the left with zeros. It understands about plus and minus signs:

>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
Basic usage of the `str.format()` method looks like this:

```python
>>> print 'We are the {} who say "{}!"'.format('knights', 'Ni')
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets refers to the position of the object passed into the `str.format()` method.

```python
>>> print '{0} and {1}'.format('spam', 'eggs')
spam and eggs
>>> print '{1} and {0}'.format('spam', 'eggs')
eggs and spam
```

If keyword arguments are used in the `str.format()` method, their values are referred to by using the name of the argument.

```python
>>> print 'This {food} is {adjective}.'.format(  
...     food='spam', adjective='absolutely horrible')
This spam is absolutely horrible.
```

Positional and keyword arguments can be arbitrarily combined:

```python
>>> print 'The story of {0}, {1}, and {other}.'.format('Bill',  
...                                                   other='Georg')
The story of Bill, Manfred, and Georg.
```

'!s' (apply `str()`) and '!r' (apply `repr()`) can be used to convert the value before it is formatted.

```python
>>> import math
>>> print 'The value of PI is approximately {}'.format(math.pi)
The value of PI is approximately 3.14159265359.
>>> print 'The value of PI is approximately {!r}'.format(math.pi)
The value of PI is approximately 3.141592653589793.
```
An optional ':.' and format specifier can follow the field name. This allows greater control over how the value is formatted. The following example rounds Pi to three places after the decimal.

```python
>>> import math
>>> print 'The value of PI is approximately {0:.3f}'.format(math.pi)
The value of PI is approximately 3.142.
```

Passing an integer after the ':.' will cause that field to be a minimum number of characters wide. This is useful for making tables pretty.

```python
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print '{0:10} ==>{1:10d}'.format(name, phone)
... Jack    ==>    4098
  Dcab    ==>    7678
  Sjoerd  ==>    4127
```

If you have a really long format string that you don’t want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets `[]` to access the keys.

```python
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print ('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '+
...     'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This could also be done by passing the table as keyword arguments with the '**' notation.

```python
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is particularly useful in combination with the built-in function
vars(), which returns a dictionary containing all local variables.

For a complete overview of string formatting with \texttt{str.format()}, see Format String Syntax.

### 7.1.1. Old string formatting

The \% operator can also be used for string formatting. It interprets the left argument much like a \texttt{sprintf()-style} format string to be applied to the right argument, and returns the string resulting from this formatting operation. For example:

```python
>>> import math
>>> print 'The value of PI is approximately %5.3f.' % math.pi
The value of PI is approximately 3.142.
```

More information can be found in the String Formatting Operations section.
open() returns a file object, and is most commonly used with two arguments: open(filename, mode).

```python
>>> f = open('workfile', 'w')
>>> print(f)
<open file 'workfile', mode 'w' at 80a960>
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. mode can be 'r' when the file will only be read, 'w' for only writing (an existing file with the same name will be erased), and 'a' opens the file for appending; any data written to the file is automatically added to the end. 'r+' opens the file for both reading and writing. The mode argument is optional; 'r' will be assumed if it's omitted.

On Windows, 'b' appended to the mode opens the file in binary mode, so there are also modes like 'rb', 'wb', and 'r+b'. Python on Windows makes a distinction between text and binary files; the end-of-line characters in text files are automatically altered slightly when data is read or written. This behind-the-scenes modification to file data is fine for ASCII text files, but it'll corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files. On Unix, it doesn't hurt to append a 'b' to the mode, so you can use it platform-independently for all binary files.

### 7.2.1. Methods of File Objects

The rest of the examples in this section will assume that a file object called `f` has already been created.
To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string. `size` is an optional numeric argument. When `size` is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most `size` bytes are read and returned. If the end of the file has been reached, `f.read()` will return an empty string (""").

```python
>>> f.read()
'This is the entire file.
' >>> f.read()
''
```

`f.readline()` reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn’t end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `''`, a string containing only a single newline.

```python
>>> f.readline()
'This is the first line of the file.
' >>> f.readline()
'Second line of the file
' >>> f.readline()
''
```

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```python
>>> for line in f:
    print line,
This is the first line of the file.
Second line of the file
```

If you want to read all the lines of a file in a list you can also use
list(f) or f.readlines().

f.write(string) writes the contents of string to the file, returning None.

```python
>>> f.write('This is a test\n')
```

To write something other than a string, it needs to be converted to a string first:

```python
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

f.tell() returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file. To change the file object's position, use f.seek(offset, from_what). The position is computed from adding offset to a reference point; the reference point is selected by the from_what argument. A from_what value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. from_what can be omitted and defaults to 0, using the beginning of the file as the reference point.

```python
>>> f = open('workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)  # Go to the 6th byte in the file
>>> f.read(1)
'd'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

When you're done with a file, call f.close() to close it and free up any system resources taken up by the open file. After calling f.close(), attempts to use the file object will automatically fail.
It is good practice to use the `with` keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent `try-finally` blocks:

```python
>>> with open('workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

File objects have some additional methods, such as `isatty()` and `truncate()` which are less frequently used; consult the Library Reference for a complete guide to file objects.

### 7.2.2. Saving structured data with `json`

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the `read()` method only returns strings, which will have to be passed to a function like `int()`, which takes a string like `'123'` and returns its numeric value 123. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called **JSON** (JavaScript Object Notation). The standard module called `json` can take Python data hierarchies, and convert them to string representations; this process is called **serializing**. Reconstructing the data from the string representation is called **deserializing**. Between serializing and
deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

**Note:** The JSON format is commonly used by modern applications to allow for data exchange. Many programmers are already familiar with it, which makes it a good choice for interoperability.

If you have an object $x$, you can view its JSON string representation with a simple line of code:

```python
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

Another variant of the `dumps()` function, called `dump()`, simply serializes the object to a file. So if $f$ is a file object opened for writing, we can do this:

```python
json.dump(x, f)
```

To decode the object again, if $f$ is a file object which has been opened for reading:

```python
x = json.load(f)
```

This simple serialization technique can handle lists and dictionaries, but serializing arbitrary class instances in JSON requires a bit of extra effort. The reference for the `json` module contains an explanation of this.

**See also:** pickle - the pickle module

Contrary to JSON, pickle is a protocol which allows the serialization of arbitrarily complex Python objects. As such, it is
specific to Python and cannot be used to communicate with applications written in other languages. It is also insecure by default: deserializing pickle data coming from an untrusted source can execute arbitrary code, if the data was crafted by a skilled attacker.
8. Errors and Exceptions

Until now error messages haven’t been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: syntax errors and exceptions.
8.1. Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print 'Hello world'
File "<stdin>", line 1
   while True print 'Hello world'
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token preceding the arrow: in the example, the error is detected at the keyword `print`, since a colon (`:`) is missing before it. File name and line number are printed so you know where to look in case the input came from a script.
8.2. Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```python
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

>>>
```python
4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
```

```python
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError` and `TypeError`. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general
it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

*Built-in Exceptions* lists the built-in exceptions and their meanings.
8.3. Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using Control-C or whatever the operating system supports); note that a user-generated interruption is signalled by raising the `KeyboardInterrupt` exception.

```python
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops! That was no valid number. Try again."
...```

The `try` statement works as follows.

- First, the `try clause` (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the `except clause` is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then if its type matches the exception named after the `except` keyword, the except clause is executed, and then execution continues after the `try` statement.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer `try` statements; if no handler is found, it is an `unhandled exception` and execution stops with a message as shown above.

A `try` statement may have more than one except clause, to specify
handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same `try` statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

```python
... except (RuntimeError, TypeError, NameError):
  ...
  pass
```

Note that the parentheses around this tuple are required, because `except ValueError, e:` was the syntax used for what is normally written as `except ValueError as e:` in modern Python (described below). The old syntax is still supported for backwards compatibility. This means `except RuntimeError, TypeError` is not equivalent to `except (RuntimeError, TypeError):` but to `except RuntimeError as TypeError:` which is not what you want.

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

```python
import sys

try:
  f = open('myfile.txt')
  s = f.readline()
  i = int(s.strip())
except IOError as e:
  print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError:
  print "Could not convert data to an integer."
except:
  print "Unexpected error!", sys.exc_info()[0]
  raise
```

The `try ... except` statement has an optional `else clause`, which,
when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

```python
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
    f.close()
```

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try ... except` statement.

When an exception occurs, it may have an associated value, also known as the exception’s `argument`. The presence and type of the argument depend on the exception type.

The except clause may specify a variable after the exception name (or tuple). The variable is bound to an exception instance with the arguments stored in `instance.args`. For convenience, the exception instance defines `__str__()` so the arguments can be printed directly without having to reference `.args`.

One may also instantiate an exception first before raising it and add any attributes to it as desired.

```python
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print type(inst)  # the exception instance
...     print inst.args  # arguments stored in .args
...     print inst  # __str__ allows args to be printed
...     x, y = inst.args
```
If an exception has an argument, it is printed as the last part ('detail') of the message for unhandled exceptions.

Exception handlers don't just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

```python
>>> def this_fails():
...     x = 1/0
... 
>>> try:
...     this_fails()
... except ZeroDivisionError as detail:
...     print 'Handling run-time error:', detail
... Handling run-time error: integer division or modulo by zero
```
8.4. Raising Exceptions

The `raise` statement allows the programmer to force a specified exception to occur. For example:

```python
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `Exception`).

If you need to determine whether an exception was raised but don’t intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

```python
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print 'An exception flew by!'
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```
8.5. User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see Classes for more about Python classes). Exceptions should typically be derived from the Exception class, either directly or indirectly. For example:

```python
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4

>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyError: 'oops!'
```

In this example, the default __init__() of Exception has been overridden. The new behavior simply creates the value attribute. This replaces the default behavior of creating the args attribute.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

```python
class Error(Exception):
    """Base class for exceptions in this module."""
```
Most exceptions are defined with names that end in “Error,” similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define. More information on classes is presented in chapter Classes.
8.6. Defining Clean-up Actions

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```python
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
... Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

A `finally` clause is always executed before leaving the `try` statement, whether an exception has occurred or not. When an exception has occurred in the `try` clause and has not been handled by an `except` clause (or it has occurred in an `except` or `else` clause), it is re-raised after the `finally` clause has been executed. The `finally` clause is also executed “on the way out” when any other clause of the `try` statement is left via a `break`, `continue` or `return` statement. A more complicated example (having `except` and `finally` clauses in the same `try` statement works as of Python 2.5):

```python
>>> def divide(x, y):
...     try:
...         result = x / y
... except ZeroDivisionError:
...     print "division by zero!"
... else:
...     print "result is", result
... finally:
...     print "executing finally clause"
... divide(2, 1)
```
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the **finally** clause is executed in any event. The
**TypeError** raised by dividing two strings is not handled by the **except**
clause and therefore re-raised after the **finally** clause has been
executed.

In real world applications, the **finally** clause is useful for releasing
external resources (such as files or network connections), regardless
of whether the use of the resource was successful.
8.7. Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```python
for line in open("myfile.txt"):
    print line,
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```python
with open("myfile.txt") as f:
    for line in f:
        print line,
```

After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines. Other objects which provide predefined clean-up actions will indicate this in their documentation.
9. Classes

Compared with other programming languages, Python’s class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are public (except see below Private Variables and Class-local References), and all member functions are virtual. As in Modula-3, there are no shorthands for referencing the object’s members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)
9.1. A Word About Names and Objects

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.
Before introducing classes, I first have to tell you something about Python’s scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what’s going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let’s begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that’s normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as `abs()`, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module’s attributes and the global names defined in the module: they share the same namespace! [1]

Attributes may be read-only or writable. In the latter case,
Assignment to attributes is possible. Module attributes are writable: you can write \texttt{modname.the\_answer = 42}. Writable attributes may also be deleted with the \texttt{del} statement. For example, \texttt{del modname.the\_answer} will remove the attribute \texttt{the\_answer} from the object named by \texttt{modname}.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called \texttt{__main__}, so they have their own global namespace. (The built-in names actually also live in a module; this is called \texttt{__builtin__}.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A \textit{scope} is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched
starting with the nearest enclosing scope, contains non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the middle scope containing the module’s global names. Otherwise, all variables found outside of the innermost scope are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don’t rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no global statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement del x removes the binding of x from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, import statements and function definitions bind the
module or function name in the local scope. (The `global` statement can be used to indicate that particular variables live in the global scope.)
9.3. A First Look at Classes

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

9.3.1. Class Definition Syntax

The simplest form of class definition looks like this:

```python
class ClassName:
    <statement-1>
    ...
    <statement-N>
```

Class definitions, like function definitions (def statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an if statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a class object is created. This is basically a wrapper around the contents of the
namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (ClassName in the example).

9.3.2. Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```python
class MyClass:
    """A simple example class""
    i = 12345

    def f(self):
        return 'hello world'
```

then MyClass.i and MyClass.f are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of MyClass.i by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: "A simple example class".

Class instantiation uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```python
x = MyClass()
```

creates a new instance of the class and assigns this object to the
local variable $x$.

The instantiation operation ("calling" a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```python
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

```python
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example,

```python
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

### 9.3.3. Instance Objects

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

`data attributes` correspond to "instance variables" in Smalltalk, and to
“data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if \( x \) is the instance of `MyClass` created above, the following piece of code will print the value 16, without leaving a trace:

```python
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

The other kind of instance attribute reference is a *method*. A method is a function that “belongs to” an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called `append`, `insert`, `remove`, `sort`, and so on. However, in the following discussion, we'll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, \( x.f \) is a valid method reference, since `MyClass.f` is a function, but \( x.i \) is not, since `MyClass.i` is not. But \( x.f \) is not the same thing as `MyClass.f` — it is a *method object*, not a function object.

### 9.3.4. Method Objects

Usually, a method is called right after it is bound:

```python
x.f()
```

In the `MyClass` example, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a
method object, and can be stored away and called at a later time. For example:

```python
xf = x.f
while True:
    print xf()
```

will continue to print `hello world` until the end of time.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn’t actually used...

Actually, you may have guessed the answer: the special thing about methods is that the object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of `n` arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method’s object before the first argument.

If you still don’t understand how methods work, a look at the implementation can perhaps clarify matters. When an instance attribute is referenced that isn't a data attribute, its class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.
9.3.5. Class and Instance Variables

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```python
class Dog:
    kind = 'canine'  # class variable shared by all instances

    def __init__(self, name):
        self.name = name  # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind
'canine'
>>> e.kind
'canine'
>>> d.name
'Fido'
>>> e.name
'Buddy'
```

As discussed in A Word About Names and Objects, shared data can have possibly surprising effects with involving mutable objects such as lists and dictionaries. For example, the `tricks` list in the following code should not be used as a class variable because just a single list would be shared by all `Dog` instances:

```python
class Dog:
    tricks = []  # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
```
Correct design of the class should use an instance variable instead:

```python
class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = []  # creates a new empty list for each

    def add_trick(self, trick):
        self.tricks.append(trick)
```

```python
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```
Data attributes override method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts. Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore), or using verbs for methods and nouns for data attributes.

Data attributes may be referenced by methods as well as by ordinary users ("clients") of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care — clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided — again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Often, the first argument of a method is called \texttt{self}. This is nothing more than a convention: the name \texttt{self} has absolutely no special meaning to Python. Note, however, that by not following the
convention your code may be less readable to other Python programmers, and it is also conceivable that a class browser program might be written that relies upon such a convention.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

h = g
```

Now f, g and h are all attributes of class C that refer to function objects, and consequently they are all methods of instances of C — h being exactly equivalent to g. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the self argument:

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```
Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing its definition. (A class is never used as a global scope.) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we'll find some good reasons why a method would want to reference its own class.

Each value is an object, and therefore has a class (also called its type). It is stored as \texttt{object.__class__}. 
9.5. Inheritance

Of course, a language feature would not be worthy of the name “class” without supporting inheritance. The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

The name `BaseClassName` must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes.
Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively virtual.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`. This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as `BaseClassName` in the global scope.)

Python has two built-in functions that work with inheritance:

- Use `isinstance()` to check an instance's type: `isinstance(obj, int)` will be True only if `obj.__class__` is `int` or some class derived from `int`.
- Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is True since `bool` is a subclass of `int`. However, `issubclass(unicode, str)` is False since `unicode` is not a subclass of `str` (they only share a common ancestor, `basestring`).

### 9.5.1. Multiple Inheritance

Python supports a limited form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```python
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```
For old-style classes, the only rule is depth-first, left-to-right. Thus, if an attribute is not found in DerivedClassName, it is searched in Base1, then (recursively) in the base classes of Base1, and only if it is not found there, it is searched in Base2, and so on.

(To some people breadth first — searching Base2 and Base3 before the base classes of Base1 — looks more natural. However, this would require you to know whether a particular attribute of Base1 is actually defined in Base1 or in one of its base classes before you can figure out the consequences of a name conflict with an attribute of Base2. The depth-first rule makes no differences between direct and inherited attributes of Base1.)

For new-style classes, the method resolution order changes dynamically to support cooperative calls to super(). This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

With new-style classes, dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all new-style classes inherit from object, so any case of multiple inheritance provides more than one path to reach object. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see https://www.python.org/download/releases/2.3/mro/.
“Private” instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. \_\_spam\_) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called name mangling. Any identifier of the form \_\_spam (at least two leading underscores, at most one trailing underscore) is textually replaced with \_classname\_spam, where classname is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

```python
class Mapping:
    def \_\_init\_(self, iterable):
        self.items_list = []
        self.\_\_update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    \_\_update = update  # private copy of original update() method

class MappingSubclass(Mapping):
```
def update(self, keys, values):
    # provides new signature for update()
    # but does not break __init__()
    for item in zip(keys, values):
        self.items_list.append(item)

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

Notice that code passed to exec, eval() or execfile() does not consider the classname of the invoking class to be the current class; this is similar to the effect of the global statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to getattr(), setattr() and delattr(), as well as when referencing __dict__ directly.
9.7. Odds and Ends

Sometimes it is useful to have a data type similar to the Pascal "record" or C "struct", bundling together a few named data items. An empty class definition will do nicely:

```python
class Employee:
    pass

john = Employee()  # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

Instance method objects have attributes, too: `m.im_self` is the instance object with the method `m()`, and `m.im_func` is the function object corresponding to the method.
User-defined exceptions are identified by classes as well. Using this mechanism it is possible to create extensible hierarchies of exceptions.

There are two new valid (semantic) forms for the `raise` statement:

```python
raise Class, instance
raise instance
```

In the first form, `instance` must be an instance of `class` or of a class derived from it. The second form is a shorthand for:

```python
raise instance.__class__, instance
```

A class in an `except` clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```python
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
```
Note that if the except clauses were reversed (with `except B` first), it would have printed B, B, B — the first matching except clause is triggered.

When an error message is printed for an unhandled exception, the exception's class name is printed, then a colon and a space, and finally the instance converted to a string using the built-in function `str()`.
9.9. Iterators

By now you have probably noticed that most container objects can be looped over using a `for` statement:

```python
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line,
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `next()` which accesses elements in the container one at a time. When there are no more elements, `next()` raises a `StopIteration` exception which tells the `for` loop to terminate. This example shows how it all works:

```python
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    it.next()
StopIteration
```
Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__()` method which returns an object with a `next()` method. If the class defines `next()`, then `__iter__()` can just return `self`:

```python
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...    print char
...
    m
    a
    p
    s
```
9.10. Generators

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the `yield` statement whenever they want to return data. Each time `next()` is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

```python
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

>>> for char in reverse('golf'):
    ...    print char
    ...
f
l
o
g
```

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `next()` methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`. In combination, these features make it easy to create iterators with no more effort than writing a regular function.
9.11. Generator Expressions

Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

Examples:

```python
>>> sum(i*i for i in range(10))  # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x, y in zip(xvec, yvec))  # dot product
260

>>> from math import pi, sin
>>> sine_table = dict((x, sin(x*pi/180)) for x in range(0, 91))

>>> unique_words = set(word for line in page for word in line.split)

>>> valedictorian = max((student.gpa, student.name) for student in data)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1,-1,-1))
['f', 'l', 'o', 'g']
```

Footnotes

[1] Except for one thing. Module objects have a secret read-only attribute called __dict__ which returns the dictionary used to implement the module’s namespace; the name __dict__ is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be
restricted to things like post-mortem debuggers.
10. Brief Tour of the Standard Library
10.1. Operating System Interface

The os module provides dozens of functions for interacting with the operating system:

```python
>>> import os
>>> os.getcwd()  # Return the current working directory
'C:\\Python26'
>>> os.chdir('/server/accesslogs')  # Change current working directory
>>> os.system('mkdir today')  # Run the command mkdir in the system shell
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```python
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```python
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```
10.2. File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```python
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```
10.3. Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance the following output results from running `python demo.py one two three` at the command line:

```python
>>> import sys
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

The `getopt` module processes `sys.argv` using the conventions of the Unix `getopt()` function. More powerful and flexible command line processing is provided by the `argparse` module.
10.4. Error Output Redirection and Program Termination

The `sys` module also has attributes for `stdin`, `stdout`, and `stderr`. The latter is useful for emitting warnings and error messages to make them visible even when `stdout` has been redirected:

```plaintext
>>> sys.stderr.write('Warning, log file not found starting a new one
Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use `sys.exit()`.
The `re` module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```python
>>> import re
>>> re.findall(r'^[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'([a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```python
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```
10.6. Mathematics

The `math` module gives access to the underlying C library functions for floating point math:

```python
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

The `random` module provides tools for making random selections:

```python
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10)  # sampling without replace
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()  # random float
0.17970987693706186
>>> random.randrange(6)  # random integer chosen from range(6)
4
```
10.7. Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are `urllib2` for retrieving data from URLs and `smtplib` for sending mail:

```python
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'
...     if 'EST' in line or 'EDT' in line: # look for Eastern
...         print line

Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org'
...     'To: jcaesar@example.org
...     From: soothsayer@example.org
...     "Beware the Ides of March."
...     '"

(Note that the second example needs a mailserver running on localhost.)
The `datetime` module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

```python
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```
10.9. Data Compression

Common data archiving and compression formats are directly supported by modules including: **zlib**, **gzip**, **bz2**, **zipfile** and **tarfile**.

```python
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```
10.10. Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The `timeit` module quickly demonstrates a modest performance advantage:

```python
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to `timeit`'s fine level of granularity, the `profile` and `pstats` modules provide tools for identifying time critical sections in larger blocks of code.
One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The doctest module provides a tool for scanning a module and validating tests embedded in a program’s docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```python
def average(values):
    """Computes the arithmetic mean of a list of numbers."
    >>> print average([20, 30, 70])
    40.0
    """
    return sum(values, 0.0) / len(values)

import doctest
doctest.testmod()  # automatically validate the embedded tests
```

The unittest module is not as effortless as the doctest module, but it allows a more comprehensive set of tests to be maintained in a separate file:

```python
import unittest
class TestStatisticalFunctions(unittest.TestCase):
    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
```
average(20, 30, 70)

unittest.main()  # Calling from the command line invokes all te
10.12. Batteries Included

Python has a “batteries included” philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The `xmlrpclib` and `SimpleXMLRPCServer` modules make implementing remote procedure calls into an almost trivial task. Despite the modules names, no direct knowledge or handling of XML is needed.
- The `email` package is a library for managing email messages, including MIME and other RFC 2822-based message documents. Unlike `smtplib` and `poplib` which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The `xml.dom` and `xml.sax` packages provide robust support for parsing this popular data interchange format. Likewise, the `csv` module supports direct reads and writes in a common database format. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- Internationalization is supported by a number of modules including `gettext`, `locale`, and the `codecs` package.
11. Brief Tour of the Standard Library – Part II

This second tour covers more advanced modules that support professional programming needs. These modules rarely occur in small scripts.
11.1. Output Formatting

The `repr` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

```python
>>> import repr
>>> repr(repr(set('supercalifragilisticexpialidocious')))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

The `pprint` module offers more sophisticated control over printing both built-in and user defined objects in a way that is readable by the interpreter. When the result is longer than one line, the “pretty printer” adds line breaks and indentation to more clearly reveal data structure:

```python
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta', 'yellow'], 'blue']]
...       [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta', 'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta', 'yellow'], 'blue']]...
```

The `textwrap` module formats paragraphs of text to fit a given screen width:

```python
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that...
...a list of strings instead of one big string with newlines t...
...the wrapped lines.""
...
>>> print textwrap.fill(doc, width=40)
The wrap() method is just like fill() except that it returns a list of strings instead of one big string with newlines
```
to separate the wrapped lines.

The `locale` module accesses a database of culture specific data formats. The grouping attribute of locale’s format function provides a direct way of formatting numbers with group separators:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()  # get a mapping of conv
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s.%f", (conv['currency_symbol'],
... conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```
11.2. Templating

The `string` module includes a versatile `Template` class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

The format uses placeholder names formed by `$` with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing `$$` creates a single escaped `$`:

```python
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

The `substitute()` method raises a `KeyError` when a placeholder is not supplied in a dictionary or a keyword argument. For mail-merge style applications, user supplied data may be incomplete and the `safe_substitute()` method may be more appropriate — it will leave placeholders unchanged if data is missing:

```python
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template subclasses can specify a custom delimiter. For example, a batch renaming utility for a photo browser may elect to use percent signs for placeholders such as the current date, image sequence number, or file format:
Another application for templating is separating program logic from the details of multiple output formats. This makes it possible to substitute custom templates for XML files, plain text reports, and HTML web reports.
11.3. Working with Binary Data Record Layouts

The `struct` module provides `pack()` and `unpack()` functions for working with variable length binary record formats. The following example shows how to loop through header information in a ZIP file without using the `zipfile` module. Pack codes "H" and "I" represent two and four byte unsigned numbers respectively. The "<" indicates that they are standard size and in little-endian byte order:

```python
import struct

data = open('myfile.zip', 'rb').read()
start = 0
for i in range(3):  # show the first 3 file
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]

    print filename, hex(crc32), comp_size, uncomp_size

    start += extra_size + comp_size  # skip to the next header
```
11.4. Multi-threading

Threading is a technique for decoupling tasks which are not sequentially dependent. Threads can be used to improve the responsiveness of applications that accept user input while other tasks run in the background. A related use case is running I/O in parallel with computations in another thread.

The following code shows how the high level `threading` module can run tasks in background while the main program continues to run:

```python
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print 'Finished background zip of:', self.infile

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print 'The main program continues to run in foreground.'

background.join()  # Wait for the background task to finish
print 'Main program waited until background was done.'
```

The principal challenge of multi-threaded applications is coordinating threads that share data or other resources. To that end, the threading module provides a number of synchronization primitives including locks, events, condition variables, and semaphores.

While those tools are powerful, minor design errors can result in
problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and then use the **Queue** module to feed that thread with requests from other threads. Applications using **Queue.Queue** objects for inter-thread communication and coordination are easier to design, more readable, and more reliable.
The `logging` module offers a full featured and flexible logging system. At its simplest, log messages are sent to a file or to `sys.stderr`:

```python
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning: config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

This produces the following output:

```
WARNING:root:Warning: config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root: Critical error -- shutting down
```

By default, informational and debugging messages are suppressed and the output is sent to standard error. Other output options include routing messages through email, datagrams, sockets, or to an HTTP Server. New filters can select different routing based on message priority: `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`.

The logging system can be configured directly from Python or can be loaded from a user editable configuration file for customized logging without altering the application.
11.6. Weak References

Python does automatic memory management (reference counting for most objects and garbage collection to eliminate cycles). The memory is freed shortly after the last reference to it has been eliminated.

This approach works fine for most applications but occasionally there is a need to track objects only as long as they are being used by something else. Unfortunately, just tracking them creates a reference that makes them permanent. The weakref module provides tools for tracking objects without creating a reference. When the object is no longer needed, it is automatically removed from a weakref table and a callback is triggered for weakref objects. Typical applications include caching objects that are expensive to create:

```python
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10) # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a # does not create a reference
>>> d['primary'] # fetch the object if it is still alive
10
>>> del a # remove the one reference
>>> gc.collect() # run garbage collection right away
0
>>> d['primary'] # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary'] # entry was automatically removed
  File "C:/python26/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```
11.7. Tools for Working with Lists

Many data structure needs can be met with the built-in list type. However, sometimes there is a need for alternative implementations with different performance trade-offs.

The `array` module provides an `array()` object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode "H") rather than the usual 16 bytes per entry for regular lists of Python int objects:

```python
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

The `collections` module provides a `deque()` object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first tree searches:

```python
>>> from collections import deque
>>> d = deque(['task1', 'task2', 'task3'])
>>> d.append('task4')
>>> print "Handling", d.popleft()
Handling task1
```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
In addition to alternative list implementations, the library also offers other tools such as the `bisect` module with functions for manipulating sorted lists:

```python
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

The `heapq` module provides functions for implementing heaps based on regular lists. The lowest valued entry is always kept at position zero. This is useful for applications which repeatedly access the smallest element but do not want to run a full list sort:

```python
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)  # rearrange the list into heap order
>>> heappush(data, -5)  # add a new entry
>>> [heappop(data) for i in range(3)]  # fetch the three smallest entries
[-5, 0, 1]
```
11.8. Decimal Floating Point Arithmetic

The `decimal` module offers a `Decimal` datatype for decimal floating point arithmetic. Compared to the built-in `float` implementation of binary floating point, the class is especially helpful for

- financial applications and other uses which require exact decimal representation,
- control over precision,
- control over rounding to meet legal or regulatory requirements,
- tracking of significant decimal places, or
- applications where the user expects the results to match calculations done by hand.

For example, calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point. The difference becomes significant if the results are rounded to the nearest cent:

```python
>>> from decimal import *
>>> x = Decimal('0.70') * Decimal('1.05')
>>> x
Decimal('0.7350')
>>> x.quantize(Decimal('0.01'))  # round to nearest cent
Decimal('0.74')
>>> round(.70 * 1.05, 2)          # same calculation with floats
0.73
```

The `Decimal` result keeps a trailing zero, automatically inferring four place significance from multiplicands with two place significance. Decimal reproduces mathematics as done by hand and avoids issues that can arise when binary floating point cannot exactly represent decimal quantities.

Exact representation enables the `Decimal` class to perform modulo
calculations and equality tests that are unsuitable for binary floating point:

```python
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995
```

```python
>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

The `decimal` module provides arithmetic with as much precision as needed:

```python
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```
12. What Now?

Reading this tutorial has probably reinforced your interest in using Python — you should be eager to apply Python to solving your real-world problems. Where should you go to learn more?

This tutorial is part of Python’s documentation set. Some other documents in the set are:

- **The Python Standard Library:**
  You should browse through this manual, which gives complete (though terse) reference material about types, functions, and the modules in the standard library. The standard Python distribution includes a *lot* of additional code. There are modules to read Unix mailboxes, retrieve documents via HTTP, generate random numbers, parse command-line options, write CGI programs, compress data, and many other tasks. Skimming through the Library Reference will give you an idea of what's available.

- **Installing Python Modules (Legacy version)** explains how to install external modules written by other Python users.

- **The Python Language Reference:** A detailed explanation of Python's syntax and semantics. It’s heavy reading, but is useful as a complete guide to the language itself.

More Python resources:

- **https://www.python.org:** The major Python Web site. It contains code, documentation, and pointers to Python-related pages around the Web. This Web site is mirrored in various places around the world, such as Europe, Japan, and Australia; a mirror may be faster than the main site, depending on your
geographical location.

- https://docs.python.org: Fast access to Python’s documentation.
- https://pypi.python.org/pypi: The Python Package Index, previously also nicknamed the Cheese Shop, is an index of user-created Python modules that are available for download. Once you begin releasing code, you can register it here so that others can find it.

For Python-related questions and problem reports, you can post to the newsgroup comp.lang.python, or send them to the mailing list at python-list@python.org. The newsgroup and mailing list are gatewayed, so messages posted to one will automatically be forwarded to the other. There are around 120 postings a day (with peaks up to several hundred), asking (and answering) questions, suggesting new features, and announcing new modules. Before posting, be sure to check the list of Frequently Asked Questions (also called the FAQ). Mailing list archives are available at https://mail.python.org/pipermail/. The FAQ answers many of the questions that come up again and again, and may already contain the solution for your problem.
13. Interactive Input Editing and History Substitution

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the GNU Readline library, which supports Emacs-style and vi-style editing. This library has its own documentation which I won't duplicate here; however, the basics are easily explained. The interactive editing and history described here are optionally available in the Unix and Cygwin versions of the interpreter.

This chapter does not document the editing facilities of Mark Hammond's PythonWin package or the Tk-based environment, IDLE, distributed with Python. The command line history recall which operates within DOS boxes on NT and some other DOS and Windows flavors is yet another beast.
13.1. Line Editing

If supported, input line editing is active whenever the interpreter prints a primary or secondary prompt. The current line can be edited using the conventional Emacs control characters. The most important of these are: `c-a` (Control-A) moves the cursor to the beginning of the line, `c-e` to the end, `c-b` moves it one position to the left, `c-f` to the right. Backspace erases the character to the left of the cursor, `c-d` the character to its right. `c-k` kills (erases) the rest of the line to the right of the cursor, `c-y` yanks back the last killed string. `c-_` undoes the last change you made; it can be repeated for cumulative effect.
13.2. History Substitution

History substitution works as follows. All non-empty input lines issued are saved in a history buffer, and when a new prompt is given you are positioned on a new line at the bottom of this buffer. \texttt{C-P} moves one line up (back) in the history buffer, \texttt{C-N} moves one down. Any line in the history buffer can be edited; an asterisk appears in front of the prompt to mark a line as modified. Pressing the \texttt{Return} key passes the current line to the interpreter. \texttt{C-R} starts an incremental reverse search; \texttt{C-S} starts a forward search.
13.3. Key Bindings

The key bindings and some other parameters of the Readline library can be customized by placing commands in an initialization file called ~/.inputrc. Key bindings have the form

key-name: function-name

or

"string": function-name

and options can be set with

set option-name value

For example:

```
# I prefer vi-style editing:
set editing-mode vi

# Edit using a single line:
set horizontal-scroll-mode On

# Rebind some keys:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Note that the default binding for Tab in Python is to insert a Tab character instead of Readline’s default filename completion function. If you insist, you can override this by putting

```
Tab: complete
```

in your ~/.inputrc. (Of course, this makes it harder to type indented
continuation lines if you’re accustomed to using Tab for that purpose.)

Automatic completion of variable and module names is optionally available. To enable it in the interpreter’s interactive mode, add the following to your startup file: [1]

```python
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

This binds the Tab key to the completion function, so hitting the Tab key twice suggests completions; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as `string.a`, it will evaluate the expression up to the final `.` and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a `__getattr__()` method is part of the expression.

A more capable startup file might look like this example. Note that this deletes the names it creates once they are no longer needed; this is done since the startup file is executed in the same namespace as the interactive commands, and removing the names avoids creating side effects in the interactive environment. You may find it convenient to keep some of the imported modules, such as `os`, which turn out to be needed in most sessions with the interpreter.

```python
# Add auto-completion and a stored history file of commands to your
# interactive interpreter. Requires Python 2.0+, readline. Auto
# bound to the Esc key by default (you can change it - see readline docs).
# Store the file in ~/.pystartup, and set an environment variable
# to it: "export PYTHONSTARTUP=~/.pystartup" in bash.

import atexit
import os
import readline
```
import rlcompleter

historyPath = os.path.expanduser("~/.pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
13.4. Alternatives to the Interactive Interpreter

This facility is an enormous step forward compared to earlier versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on continuation lines (the parser knows if an indent token is required next). The completion mechanism might use the interpreter’s symbol table. A command to check (or even suggest) matching parentheses, quotes, etc., would also be useful.

One alternative enhanced interactive interpreter that has been around for quite some time is IPython, which features tab completion, object exploration and advanced history management. It can also be thoroughly customized and embedded into other applications. Another similar enhanced interactive environment is bpython.

Footnotes

[1] Python will execute the contents of a file identified by the PYTHONSTARTUP environment variable when you start an interactive interpreter. To customize Python even for non-interactive mode, see The Customization Modules.
14. Floating Point Arithmetic: Issues and Limitations

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the decimal fraction

0.125

has value $1/10 + 2/100 + 5/1000$, and in the same way the binary fraction

0.001

has value $0/2 + 0/4 + 1/8$. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.

The problem is easier to understand at first in base 10. Consider the fraction $1/3$. You can approximate that as a base 10 fraction:

0.3

or, better,

0.33

or, better,

0.333
and so on. No matter how many digits you're willing to write down, the result will never be exactly $1/3$, but will be an increasingly better approximation of $1/3$.

In the same way, no matter how many base 2 digits you're willing to use, the decimal value 0.1 cannot be represented exactly as a base 2 fraction. In base 2, $1/10$ is the infinitely repeating fraction

0.00011001100110011001100110011001100110011001100110011001100110011001100110011001100110011001100110011001100110011001100110011001...

Stop at any finite number of bits, and you get an approximation.

On a typical machine running Python, there are 53 bits of precision available for a Python float, so the value stored internally when you enter the decimal number 0.1 is the binary fraction

0.00011001100110011001100110011001100110011001100110011010

which is close to, but not exactly equal to, $1/10$.

It's easy to forget that the stored value is an approximation to the original decimal fraction, because of the way that floats are displayed at the interpreter prompt. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. If Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display

```
>>> 0.1
0.100000000000000055511151231257827021181583404541015625
```

That is more digits than most people find useful, so Python keeps the number of digits manageable by displaying a rounded value instead

```
>>> 0.1
0.1
```
It’s important to realize that this is, in a real sense, an illusion: the value in the machine is not exactly $1/10$, you’re simply rounding the display of the true machine value. This fact becomes apparent as soon as you try to do arithmetic with these values

```
>>> 0.1 + 0.2
0.30000000000000004
```

Note that this is in the very nature of binary floating-point: this is not a bug in Python, and it is not a bug in your code either. You’ll see the same kind of thing in all languages that support your hardware’s floating-point arithmetic (although some languages may not display the difference by default, or in all output modes).

Other surprises follow from this one. For example, if you try to round the value $2.675$ to two decimal places, you get this

```
>>> round(2.675, 2)
2.67
```

The documentation for the built-in `round()` function says that it rounds to the nearest value, rounding ties away from zero. Since the decimal fraction $2.675$ is exactly halfway between $2.67$ and $2.68$, you might expect the result here to be (a binary approximation to) $2.68$. It’s not, because when the decimal string $2.675$ is converted to a binary floating-point number, it’s again replaced with a binary approximation, whose exact value is

```
2.67499999999982236431605997495353221893310546875
```

Since this approximation is slightly closer to $2.67$ than to $2.68$, it’s rounded down.

If you’re in a situation where you care which way your decimal halfway-cases are rounded, you should consider using the `decimal`
module. Incidentally, the `decimal` module also provides a nice way to "see" the exact value that's stored in any particular Python float

```python
>>> from decimal import Decimal
>>> Decimal(2.675)
Decimal('2.67499999999999982236431605997495353221893310546875')
```

Another consequence is that since 0.1 is not exactly 1/10, summing ten values of 0.1 may not yield exactly 1.0, either:

```python
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.9999999999999999
```

Binary floating-point arithmetic holds many surprises like this. The problem with “0.1” is explained in precise detail below, in the “Representation Error” section. See The Perils of Floating Point for a more complete account of other common surprises.

As that says near the end, “there are no easy answers.” Still, don't be unduly wary of floating-point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in $2^{53}$ per operation. That's more than adequate for most tasks, but you do need to keep in mind that it's not decimal arithmetic, and that every float operation can suffer a new rounding error.

While pathological cases do exist, for most casual use of floating-point arithmetic you'll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. For fine control over how a float is displayed see the `str.format()` method's format specifiers in Format String Syntax.
14.1. Representation Error

This section explains the “0.1” example in detail, and shows how you can perform an exact analysis of cases like this yourself. Basic familiarity with binary floating-point representation is assumed.

*Representation error* refers to the fact that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won’t display the exact decimal number you expect:

```
>>> 0.1 + 0.2
0.30000000000000004
```

Why is that? 1/10 and 2/10 are not exactly representable as a binary fraction. Almost all machines today (July 2010) use IEEE-754 floating point arithmetic, and almost all platforms map Python floats to IEEE-754 “double precision”. 754 doubles contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form $J/2^N$ where $J$ is an integer containing exactly 53 bits. Rewriting

```
1 / 10 ~= J / (2**N)
```

as

```
J ~= 2**N / 10
```

and recalling that $J$ has exactly 53 bits (is $>= 2^{52}$ but $< 2^{53}$), the best value for $N$ is 56:

```
>>> 2**52
4503599627370496
>>> 2**53
```
That is, 56 is the only value for \( N \) that leaves \( J \) with exactly 53 bits. The best possible value for \( J \) is then that quotient rounded:

\[
q, \ r = \text{divmod}(2^{56}, 10)
\]

Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

\[
q + 1
\]

Therefore the best possible approximation to \( 1/10 \) in 754 double precision is that over \( 2^{56} \), or

\[
7205759403792794 \div 72057594037927936
\]

Note that since we rounded up, this is actually a little bit larger than \( 1/10 \); if we had not rounded up, the quotient would have been a little bit smaller than \( 1/10 \). But in no case can it be \textit{exactly} \( 1/10 \)!

So the computer never “sees” \( 1/10 \): what it sees is the exact fraction given above, the best 754 double approximation it can get:

\[
.1 \times 2^{56}
\]

If we multiply that fraction by \( 10^{30} \), we can see the (truncated) value of its 30 most significant decimal digits:
meaning that the exact number stored in the computer is approximately equal to the decimal value 0.100000000000000005551115123125. In versions prior to Python 2.7 and Python 3.1, Python rounded this value to 17 significant digits, giving ‘0.10000000000000001’. In current versions, Python displays a value based on the shortest decimal fraction that rounds correctly back to the true binary value, resulting simply in ‘0.1’.
15. Appendix
15.1. Interactive Mode

15.1.1. Error Handling

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace. (Exceptions handled by an except clause in a try statement are not errors in this context.) Some errors are unconditionally fatal and cause an exit with a nonzero exit; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to the standard error stream; normal output from executed commands is written to standard output.

Typing the interrupt character (usually Control-C or Delete) to the primary or secondary prompt cancels the input and returns to the primary prompt. [1] Typing an interrupt while a command is executing raises the KeyboardInterrupt exception, which may be handled by a try statement.

15.1.2. Executable Python Scripts

On BSD’ish Unix systems, Python scripts can be made directly executable, like shell scripts, by putting the line

```
#!/usr/bin/env python
```

(assuming that the interpreter is on the user’s PATH) at the beginning of the script and giving the file an executable mode. The #! must be the first two characters of the file. On some platforms, this first line must end with a Unix-style line ending (‘\n’), not a Windows
The script can be given an executable mode, or permission, using the *chmod* command.

```
$ chmod +x myscript.py
```

On Windows systems, there is no notion of an “executable mode”. The Python installer automatically associates `.py` files with *python.exe* so that a double-click on a Python file will run it as a script. The extension can also be `.pyw`, in that case, the console window that normally appears is suppressed.

### 15.1.3. The Interactive Startup File

When you use Python interactively, it is frequently handy to have some standard commands executed every time the interpreter is started. You can do this by setting an environment variable named `PYTHONSTARTUP` to the name of a file containing your start-up commands. This is similar to the `.profile` feature of the Unix shells.

This file is only read in interactive sessions, not when Python reads commands from a script, and not when `/dev/tty` is given as the explicit source of commands (which otherwise behaves like an interactive session). It is executed in the same namespace where interactive commands are executed, so that objects that it defines or imports can be used without qualification in the interactive session. You can also change the prompts `sys.ps1` and `sys.ps2` in this file.

If you want to read an additional start-up file from the current directory, you can program this in the global start-up file using code like

```python
if os.path.isfile('.pythonrc.py'):
    exec(open('.pythonrc.py').read())
```

If you want to use the startup
file in a script, you must do this explicitly in the script:

```python
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

### 15.1.4. The Customization Modules

Python provides two hooks to let you customize it: `sitecustomize` and `usercustomize`. To see how it works, you need first to find the location of your user site-packages directory. Start Python and run this code:

```python
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python2.7/site-packages'
```

Now you can create a file named `usercustomize.py` in that directory and put anything you want in it. It will affect every invocation of Python, unless it is started with the `-s` option to disable the automatic import.

`sitecustomize` works in the same way, but is typically created by an administrator of the computer in the global site-packages directory, and is imported before `usercustomize`. See the documentation of the `site` module for more details.

### Footnotes

[1] A problem with the GNU Readline package may prevent this.
1. Command line and environment

The CPython interpreter scans the command line and the environment for various settings.

**CPython implementation detail:** Other implementations’ command line schemes may differ. See Alternate Implementations for further resources.
1.1. Command line

When invoking Python, you may specify any of these options:

```
python [-BdEi0QsRStuUvVwxX3?] [-c command | -m module-name | script]
```

The most common use case is, of course, a simple invocation of a script:

```
python myscript.py
```

1.1.1. Interface options

The interpreter interface resembles that of the UNIX shell, but provides some additional methods of invocation:

- When called with standard input connected to a tty device, it prompts for commands and executes them until an EOF (an end-of-file character, you can produce that with `Ctrl-D` on UNIX or `Ctrl-Z`, `Enter` on Windows) is read.
- When called with a file name argument or with a file as standard input, it reads and executes a script from that file.
- When called with a directory name argument, it reads and executes an appropriately named script from that directory.
- When called with `-c command`, it executes the Python statement(s) given as `command`. Here `command` may contain multiple statements separated by newlines. Leading whitespace is significant in Python statements!
- When called with `-m module-name`, the given module is located on the Python module path and executed as a script.

In non-interactive mode, the entire input is parsed before it is
executed.

An interface option terminates the list of options consumed by the interpreter, all consecutive arguments will end up in `sys.argv`—note that the first element, subscript zero (`sys.argv[0]`), is a string reflecting the program's source.

- **c** `<command>`
  
  Execute the Python code in `command`. `command` can be one or more statements separated by newlines, with significant leading whitespace as in normal module code.

  If this option is given, the first element of `sys.argv` will be "-c" and the current directory will be added to the start of `sys.path` (allowing modules in that directory to be imported as top level modules).

- **m** `<module-name>`
  
  Search `sys.path` for the named module and execute its contents as the `__main__` module.

  Since the argument is a `module` name, you must not give a file extension (.py). The `module-name` should be a valid Python module name, but the implementation may not always enforce this (e.g. it may allow you to use a name that includes a hyphen).

  Package names are also permitted. When a package name is supplied instead of a normal module, the interpreter will execute `<pkg>.__main__` as the main module. This behaviour is deliberately similar to the handling of directories and zipfiles that are passed to the interpreter as the script argument.

  **Note:** This option cannot be used with built-in modules and extension modules written in C, since they do not have Python module files. However, it can still be used for precompiled
modules, even if the original source file is not available.

If this option is given, the first element of `sys.argv` will be the full path to the module file. As with the `-c` option, the current directory will be added to the start of `sys.path`

Many standard library modules contain code that is invoked on their execution as a script. An example is the `timeit` module:

```bash
python -mtimeit -s 'setup here' 'benchmarked code here'
python -mtimeit -h # for details
```

See also:

`runpy.run_module()`

Equivalent functionality directly available to Python code

**PEP 338** – Executing modules as scripts

*New in version 2.4.*

*Changed in version 2.5:* The named module can now be located inside a package.

*Changed in version 2.7:* Supply the package name to run a `__main__` submodule. `sys.argv[0]` is now set to "-m" while searching for the module (it was previously incorrectly set to "-c")

- Read commands from standard input (`sys.stdin`). If standard input is a terminal, `-i` is implied.

If this option is given, the first element of `sys.argv` will be "-m" and the current directory will be added to the start of `sys.path`.

-
<script>
Execute the Python code contained in script, which must be a filesystem path (absolute or relative) referring to either a Python file, a directory containing a __main__.py file, or a zipfile containing a __main__.py file.

If this option is given, the first element of sys.argv will be the script name as given on the command line.

If the script name refers directly to a Python file, the directory containing that file is added to the start of sys.path, and the file is executed as the __main__ module.

If the script name refers to a directory or zipfile, the script name is added to the start of sys.path and the __main__.py file in that location is executed as the __main__ module.

Changed in version 2.5: Directories and zipfiles containing a __main__.py file at the top level are now considered valid Python scripts.

If no interface option is given, -i is implied, sys.argv[0] is an empty string ("") and the current directory will be added to the start of sys.path.

See also: Invoking the Interpreter

1.1.2. Generic options
```
-?
-h
--help
    Print a short description of all command line options.

    Changed in version 2.5: The --help variant.

-V
--version
    Print the Python version number and exit. Example output could be:

    Python 2.5.1

    Changed in version 2.5: The --version variant.

1.1.3. Miscellaneous options

-B
    If given, Python won't try to write .pyc or .pyo files on the import
    of source modules. See also PYTHONDON'TWRITEBYTECODE.

    New in version 2.6.

-d
    Turn on parser debugging output (for wizards only, depending on
    compilation options). See also PYTHONDEBUG.

-E
    Ignore all PYTHON* environment variables, e.g. PYTHONPATH and
    PYTHONHOME, that might be set.

    New in version 2.2.

-i
    When a script is passed as first argument or the -c option is
used, enter interactive mode after executing the script or the command, even when `sys.stdin` does not appear to be a terminal. The `PYTHONSTARTUP` file is not read.

This can be useful to inspect global variables or a stack trace when a script raises an exception. See also `PYTHONINSPECT`.

- **-O**
  Turn on basic optimizations. This changes the filename extension for compiled (bytecode) files from `.pyc` to `.pyo`. See also `PYTHONOPTIMIZE`.

- **-OO**
  Discard docstrings in addition to the `-O` optimizations.

- **-Q <arg>**
  Division control. The argument must be one of the following:

  - **old**
    division of int/int and long/long return an int or long (**default**)
  
    - **new**
      new division semantics, i.e. division of int/int and long/long returns a float
  
    - **warn**
      old division semantics with a warning for int/int and long/long
  
    - **warnall**
      old division semantics with a warning for all uses of the division operator

See also: `Tools/scripts/fixdiv.py` for a use of `warnall`

**PEP 238** – Changing the division operator
Turn on hash randomization, so that the `__hash__()` values of str, bytes and datetime objects are “salted” with an unpredictable random value. Although they remain constant within an individual Python process, they are not predictable between repeated invocations of Python.

This is intended to provide protection against a denial-of-service caused by carefully-chosen inputs that exploit the worst case performance of a dict construction, $O(n^2)$ complexity. See http://www.ocert.org/advisories/ocert-2011-003.html for details.

Changing hash values affects the order in which keys are retrieved from a dict. Although Python has never made guarantees about this ordering (and it typically varies between 32-bit and 64-bit builds), enough real-world code implicitly relies on this non-guaranteed behavior that the randomization is disabled by default.

See also `PYTHONHASHSEED`.

New in version 2.6.8.

Don’t add the user site-packages directory to `sys.path`.

New in version 2.6.

See also: PEP 370 – Per user site-packages directory

Disable the import of the module `site` and the site-dependent manipulations of `sys.path` that it entails.
-t
Issue a warning when a source file mixes tabs and spaces for indentation in a way that makes it depend on the worth of a tab expressed in spaces. Issue an error when the option is given twice (-tt).

-u
Force stdin, stdout and stderr to be totally unbuffered. On systems where it matters, also put stdin, stdout and stderr in binary mode.

Note that there is internal buffering in file.readlines() and File Objects (for line in sys.stdin) which is not influenced by this option. To work around this, you will want to use file.readline() inside a while 1: loop.

See also PYTHONUNBUFFERED.

-v
Print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded. When given twice (-vv), print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit. See also PYTHONVERBOSE.

-W arg
Warning control. Python's warning machinery by default prints warning messages to sys.stderr. A typical warning message has the following form:

```plaintext
file:line: category: message
```

By default, each warning is printed once for each source line where it occurs. This option controls how often warnings are printed.
Multiple `-w` options may be given; when a warning matches more than one option, the action for the last matching option is performed. Invalid `-w` options are ignored (though, a warning message is printed about invalid options when the first warning is issued).

Starting from Python 2.7, `DeprecationWarning` and its descendants are ignored by default. The `-wd` option can be used to re-enable them.

Warnings can also be controlled from within a Python program using the `warnings` module.

The simplest form of argument is one of the following action strings (or a unique abbreviation) by themselves:

- **ignore**
  
  Ignore all warnings.

- **default**
  
  Explicitly request the default behavior (printing each warning once per source line).

- **all**
  
  Print a warning each time it occurs (this may generate many messages if a warning is triggered repeatedly for the same source line, such as inside a loop).

- **module**
  
  Print each warning only the first time it occurs in each module.

- **once**
  
  Print each warning only the first time it occurs in the program.

- **error**
  
  Raise an exception instead of printing a warning message.
The full form of argument is:

```
action:message:category:module:line
```

Here, `action` is as explained above but only applies to messages that match the remaining fields. Empty fields match all values; trailing empty fields may be omitted. The `message` field matches the start of the warning message printed; this match is case-insensitive. The `category` field matches the warning category. This must be a class name; the match tests whether the actual warning category of the message is a subclass of the specified warning category. The full class name must be given. The `module` field matches the (fully-qualified) module name; this match is case-sensitive. The `line` field matches the line number, where zero matches all line numbers and is thus equivalent to an omitted line number.

**See also:** warnings – the warnings module  
PEP 230 – Warning framework  
PYTHONWARNINGS

**-x**

Skip the first line of the source, allowing use of non-Unix forms of `#!cmd`. This is intended for a DOS specific hack only.

**Note:** The line numbers in error messages will be off by one.

**-3**

Warn about Python 3.x possible incompatibilities by emitting a `DeprecationWarning` for features that are removed or significantly changed in Python 3.

*New in version 2.6.*
1.1.4. Options you shouldn’t use

- **J**
  
  Reserved for use by Jython.

- **U**
  
  Turns all string literals into unicodes globally. Do not be tempted to use this option as it will probably break your world. It also produces `.pyc` files with a different magic number than normal. Instead, you can enable unicode literals on a per-module basis by using:

  ```python
  from __future__ import unicode_literals
  ```

  at the top of the file. See `__future__` for details.

- **X**
  
  Reserved for alternative implementations of Python to use for their own purposes.
1.2. Environment variables

These environment variables influence Python’s behavior, they are processed before the command-line switches other than -E. It is customary that command-line switches override environmental variables where there is a conflict.

**PYTHONHOME**

Change the location of the standard Python libraries. By default, the libraries are searched in `prefix/lib/pythonversion` and `exec_prefix/lib/pythonversion`, where `prefix` and `exec_prefix` are installation-dependent directories, both defaulting to `/usr/local`.

When `PYTHONHOME` is set to a single directory, its value replaces both `prefix` and `exec_prefix`. To specify different values for these, set `PYTHONHOME` to `prefix:exec_prefix`.

**PYTHONPATH**

Augment the default search path for module files. The format is the same as the shell’s `PATH`: one or more directory pathnames separated by `os.pathsep` (e.g. colons on Unix or semicolons on Windows). Non-existent directories are silently ignored.

In addition to normal directories, individual `PYTHONPATH` entries may refer to zipfiles containing pure Python modules (in either source or compiled form). Extension modules cannot be imported from zipfiles.

The default search path is installation dependent, but generally begins with `prefix/lib/pythonversion` (see `PYTHONHOME` above). It is *always* appended to `PYTHONPATH`. 
An additional directory will be inserted in the search path in front of \texttt{PYTHONPATH} as described above under \textit{Interface options}. The search path can be manipulated from within a Python program as the variable \texttt{sys.path}.

\textbf{PYTHONSTARTUP}

If this is the name of a readable file, the Python commands in that file are executed before the first prompt is displayed in interactive mode. The file is executed in the same namespace where interactive commands are executed so that objects defined or imported in it can be used without qualification in the interactive session. You can also change the prompts \texttt{sys.ps1} and \texttt{sys.ps2} in this file.

\textbf{PYTHONY2K}

Set this to a non-empty string to cause the \texttt{time} module to require dates specified as strings to include 4-digit years, otherwise 2-digit years are converted based on rules described in the \texttt{time} module documentation.

\textbf{PYTHONOPTIMIZE}

If this is set to a non-empty string it is equivalent to specifying the \texttt{-O} option. If set to an integer, it is equivalent to specifying \texttt{-O} multiple times.

\textbf{PYTHONDEBUG}

If this is set to a non-empty string it is equivalent to specifying the \texttt{-d} option. If set to an integer, it is equivalent to specifying \texttt{-d} multiple times.

\textbf{PYTHONINSPECT}

If this is set to a non-empty string it is equivalent to specifying the \texttt{-i} option.

This variable can also be modified by Python code using
os.environ to force inspect mode on program termination.

**PYTHONUNBUFFERED**
If this is set to a non-empty string it is equivalent to specifying the 
-u option.

**PYTHONVERBOSE**
If this is set to a non-empty string it is equivalent to specifying the 
-v option. If set to an integer, it is equivalent to specifying -v 
multiple times.

**PYTHONCASEOK**
If this is set, Python ignores case in import statements. This only 
works on Windows, OS X, OS/2, and RiscOS.

**PYTHONDONTWRITEBYTECODE**
If this is set, Python won’t try to write .pyc or .pyo files on the 
import of source modules. This is equivalent to specifying the -B 
option.

*New in version 2.6.*

**PYTHONHASHSEED**
If this variable is set to random, the effect is the same as 
specifying the -R option: a random value is used to seed the 
hashes of str, bytes and datetime objects.

If PYTHONHASHSEED is set to an integer value, it is used as a fixed 
seed for generating the hash() of the types covered by the hash 
randomization.

Its purpose is to allow repeatable hashing, such as for selftests 
for the interpreter itself, or to allow a cluster of python processes 
to share hash values.

The integer must be a decimal number in the range
[0,4294967295]. Specifying the value 0 will lead to the same hash values as when hash randomization is disabled.

New in version 2.6.8.

**PYTHONIOENCODING**

Overrides the encoding used for stdin/stdout/stderr, in the syntax `encodingname:errorhandler`. The `:errorhandler` part is optional and has the same meaning as in `str.encode()`.

New in version 2.6.

**PYTHONNOUSERSITE**

If this is set, Python won’t add the user site-packages directory to `sys.path`.

New in version 2.6.

See also: PEP 370 – Per user site-packages directory

**PYTHONUSERBASE**

Defines the user base directory, which is used to compute the path of the user site-packages directory and Distutils installation paths for `python setup.py install --user`.

New in version 2.6.

See also: PEP 370 – Per user site-packages directory

**PYTHONEXECUTABLE**

If this environment variable is set, `sys.argv[0]` will be set to its value instead of the value got through the C runtime. Only works on Mac OS X.

**PYTHONWARNINGS**
This is equivalent to the \texttt{-w} option. If set to a comma separated string, it is equivalent to specifying \texttt{-w} multiple times.

\textbf{PYTHONHTTPSVERIFY}

If this environment variable is set specifically to \texttt{0}, then it is equivalent to implicitly calling \texttt{ssl\_https\_verify\_certificates()} with \texttt{enable=False} when \texttt{ssl} is first imported.

Refer to the documentation of \texttt{ssl\_https\_verify\_certificates()} for details.

\textit{New in version 2.7.12.}

\section*{1.2.1. Debug-mode variables}

Setting these variables only has an effect in a debug build of Python, that is, if Python was configured with the \texttt{--with-pydebug} build option.

\textbf{PYTHONTHREADDEBUG}

If set, Python will print threading debug info.

\textit{Changed in version 2.6:} Previously, this variable was called \texttt{THREADDEBUG}.

\textbf{PYTHONDUMPREFS}

If set, Python will dump objects and reference counts still alive after shutting down the interpreter.

\textbf{PYTHONMALLOCSTATS}

If set, Python will print memory allocation statistics every time a new object arena is created, and on shutdown.
2. Using Python on Unix platforms
2.1. Getting and installing the latest version of Python

2.1.1. On Linux

Python comes preinstalled on most Linux distributions, and is available as a package on all others. However there are certain features you might want to use that are not available on your distro’s package. You can easily compile the latest version of Python from source.

In the event that Python doesn’t come preinstalled and isn’t in the repositories as well, you can easily make packages for your own distro. Have a look at the following links:

See also:

   for Debian users

https://en.opensuse.org/Portal:Packaging
   for OpenSuse users

   for Fedora users

   for Slackware users

2.1.2. On FreeBSD and OpenBSD
• FreeBSD users, to add the package use:

```bash
pkg_add -r python
```

• OpenBSD users use:

```bash
```

For example i386 users get the 2.5.1 version of Python using:

```bash
pkg_add ftp://ftp.openbsd.org/pub/OpenBSD/4.2/packages/i386/python-2.5.1p2.tgz
```

2.1.3. On OpenSolaris

You can get Python from OpenCSW. Various versions of Python are available and can be installed with e.g. `pkgutil -i python27`.
### 2.2. Building Python

If you want to compile CPython yourself, first thing you should do is get the source. You can download either the latest release’s source or just grab a fresh clone. (If you want to contribute patches, you will need a clone.)

The build process consists in the usual

```
./configure
make
make install
```

invocations. Configuration options and caveats for specific Unix platforms are extensively documented in the README file in the root of the Python source tree.

**Warning:** `make install` can overwrite or masquerade the `python` binary. `make altinstall` is therefore recommended instead of `make install` since it only installs `exec_prefix/bin/pythonversion`. 
2.3. Python-related paths and files

These are subject to difference depending on local installation conventions; `prefix (${prefix})` and `exec_prefix (${exec_prefix})` are installation-dependent and should be interpreted as for GNU software; they may be the same.

For example, on most Linux systems, the default for both is `/usr`.

<table>
<thead>
<tr>
<th>File/directory</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exec_prefix/bin/python</code></td>
<td>Recommended location of the interpreter.</td>
</tr>
<tr>
<td><code>prefix/lib/pythonversion</code>, <code>exec_prefix/lib/pythonversion</code></td>
<td>Recommended locations of the directories containing the standard modules.</td>
</tr>
<tr>
<td><code>prefix/include/pythonversion</code>, <code>exec_prefix/include/pythonversion</code></td>
<td>Recommended locations of the directories containing the include files needed for developing Python extensions and embedding the interpreter.</td>
</tr>
<tr>
<td><code>~/.pythonrc.py</code></td>
<td>User-specific initialization file loaded by the user module; not used by default or by most applications.</td>
</tr>
</tbody>
</table>
2.4. Miscellaneous

To easily use Python scripts on Unix, you need to make them executable, e.g. with

```
$ chmod +x script
```

and put an appropriate Shebang line at the top of the script. A good choice is usually

```
#!/usr/bin/env python
```

which searches for the Python interpreter in the whole PATH. However, some Unices may not have the env command, so you may need to hardcode `/usr/bin/python` as the interpreter path.

To use shell commands in your Python scripts, look at the subprocess module.
2.5. Editors

Vim and Emacs are excellent editors which support Python very well. For more information on how to code in Python in these editors, look at:

- https://sourceforge.net/projects/python-mode

Geany is an excellent IDE with support for a lot of languages. For more information, read: https://www.geany.org/

Komodo edit is another extremely good IDE. It also has support for a lot of languages. For more information, read https://komodoide.com/.
3. Using Python on Windows

This document aims to give an overview of Windows-specific behaviour you should know about when using Python on Microsoft Windows.
3.1. Installing Python

Unlike most Unix systems and services, Windows does not require Python natively and thus does not pre-install a version of Python. However, the CPython team has compiled Windows installers (MSI packages) with every release for many years.

With ongoing development of Python, some platforms that used to be supported earlier are no longer supported (due to the lack of users or developers). Check PEP 11 for details on all unsupported platforms.

- DOS and Windows 3.x are deprecated since Python 2.0 and code specific to these systems was removed in Python 2.1.
- Up to 2.5, Python was still compatible with Windows 95, 98 and ME (but already raised a deprecation warning on installation). For Python 2.6 (and all following releases), this support was dropped and new releases are just expected to work on the Windows NT family.
- Windows CE is still supported.
- The Cygwin installer offers to install the Python interpreter as well (cf. Cygwin package source, Maintainer releases)

See Python for Windows (and DOS) for detailed information about platforms with precompiled installers.

See also:

**Python on XP**
“7 Minutes to “Hello World!”” by Richard Dooling, 2006

**Installing on Windows**
For Windows users

3.2. Alternative bundles

Besides the standard CPython distribution, there are modified packages including additional functionality. The following is a list of popular versions and their key features:

ActivePython
   Installer with multi-platform compatibility, documentation,
   PyWin32

Enthought Python Distribution
   Popular modules (such as PyWin32) with their respective documentation, tool suite for building extensible Python applications

Notice that these packages are likely to install *older* versions of Python.
3.3. Configuring Python

In order to run Python flawlessly, you might have to change certain environment settings in Windows.

3.3.1. Excursus: Setting environment variables

Windows has a built-in dialog for changing environment variables (following guide applies to XP classical view): Right-click the icon for your machine (usually located on your Desktop and called “My Computer”) and choose Properties there. Then, open the Advanced tab and click the Environment Variables button.

In short, your path is:

My Computer ➤ Properties ➤ Advanced ➤ Environment Variables

In this dialog, you can add or modify User and System variables. To change System variables, you need non-restricted access to your machine (i.e. Administrator rights).

Another way of adding variables to your environment is using the `set` command:

```bash
set PYTHONPATH=%PYTHONPATH%;C:\My_python_lib
```

To make this setting permanent, you could add the corresponding command line to your `autoexec.bat`. `msconfig` is a graphical interface to this file.

Viewing environment variables can also be done more straightforward: The command prompt will expand strings wrapped into percent signs automatically:
Consult `set ?` for details on this behaviour.

**See also:**

- [https://support.microsoft.com/kb/100843](https://support.microsoft.com/kb/100843)  
  Environment variables in Windows NT
- [https://support.microsoft.com/kb/310519](https://support.microsoft.com/kb/310519)  
  How To Manage Environment Variables in Windows XP
- [https://www.chem.gla.ac.uk/~louis/software/faq/q1.html](https://www.chem.gla.ac.uk/~louis/software/faq/q1.html)  
  Setting Environment variables, Louis J. Farrugia

### 3.3.2. Finding the Python executable

Besides using the automatically created start menu entry for the Python interpreter, you might want to start Python in the DOS prompt. To make this work, you need to set your `%PATH%` environment variable to include the directory of your Python distribution, delimited by a semicolon from other entries. An example variable could look like this (assuming the first two entries are Windows’ default):

```
C:\WINDOWS\system32;C:\WINDOWS;C:\Python25
```

Typing `python` on your command prompt will now fire up the Python interpreter. Thus, you can also execute your scripts with command line options, see [Command line documentation](https://docs.python.org/3/library/cmd.html).

### 3.3.3. Finding modules

Python usually stores its library (and thereby your site-packages folder) in the installation directory. So, if you had installed Python to `C:\Python\`, the default library would reside in `C:\Python\Lib` and
third-party modules should be stored in C:\Python\Lib\site-packages.

This is how `sys.path` is populated on Windows:

- An empty entry is added at the start, which corresponds to the current directory.
- If the environment variable `PYTHONPATH` exists, as described in Environment variables, its entries are added next. Note that on Windows, paths in this variable must be separated by semicolons, to distinguish them from the colon used in drive identifiers (c: \ etc.).
- Additional “application paths” can be added in the registry as subkeys of `\SOFTWARE\Python\PythonCore\version\PythonPath` under both the `HKEY_CURRENT_USER` and `HKEY_LOCAL_MACHINE` hives. Subkeys which have semicolon-delimited path strings as their default value will cause each path to be added to `sys.path`. (Note that all known installers only use HKLM, so HKCU is typically empty.)
- If the environment variable `PYTHONHOME` is set, it is assumed as “Python Home”. Otherwise, the path of the main Python executable is used to locate a “landmark file” (Lib\os.py) to deduce the “Python Home”. If a Python home is found, the relevant sub-directories added to `sys.path` (Lib, plat-win, etc) are based on that folder. Otherwise, the core Python path is constructed from the PythonPath stored in the registry.
- If the Python Home cannot be located, no `PYTHONPATH` is specified in the environment, and no registry entries can be found, a default path with relative entries is used (e.g. .\Lib; .\plat-win, etc).

The end result of all this is:
• When running python.exe, or any other .exe in the main Python directory (either an installed version, or directly from the PCbuild directory), the core path is deduced, and the core paths in the registry are ignored. Other “application paths” in the registry are always read.

• When Python is hosted in another .exe (different directory, embedded via COM, etc), the “Python Home” will not be deduced, so the core path from the registry is used. Other “application paths” in the registry are always read.

• If Python can’t find its home and there is no registry (eg, frozen .exe, some very strange installation setup) you get a path with some default, but relative, paths.

### 3.3.4. Executing scripts

Python scripts (files with the extension .py) will be executed by python.exe by default. This executable opens a terminal, which stays open even if the program uses a GUI. If you do not want this to happen, use the extension .pyw which will cause the script to be executed by pythonw.exe by default (both executables are located in the top-level of your Python installation directory). This suppresses the terminal window on startup.

You can also make all .py scripts execute with pythonw.exe, setting this through the usual facilities, for example (might require administrative rights):

1. Launch a command prompt.

2. Associate the correct file group with .py scripts:

   ```bash
   assoc .py=Python.File
   ```

3. Redirect all Python files to the new executable:
ftype Python.File=C:\Path\to\pythonw.exe "%1" %*
3.4. Additional modules

Even though Python aims to be portable among all platforms, there are features that are unique to Windows. A couple of modules, both in the standard library and external, and snippets exist to use these features.

The Windows-specific standard modules are documented in MS Windows Specific Services.

3.4.1. PyWin32

The PyWin32 module by Mark Hammond is a collection of modules for advanced Windows-specific support. This includes utilities for:

- Component Object Model (COM)
- Win32 API calls
- Registry
- Event log
- Microsoft Foundation Classes (MFC) user interfaces

PythonWin is a sample MFC application shipped with PyWin32. It is an embeddable IDE with a built-in debugger.

See also:

Win32 How Do I...?
  by Tim Golden

Python and COM
  by David and Paul Boddie

3.4.2. Py2exe
**Py2exe** is a **distutils** extension (see Extending Distutils) which wraps Python scripts into executable Windows programs (*.exe files). When you have done this, you can distribute your application without requiring your users to install Python.

### 3.4.3. WConio

Since Python’s advanced terminal handling layer, **curses**, is restricted to Unix-like systems, there is a library exclusive to Windows as well: Windows Console I/O for Python.

**WConio** is a wrapper for Turbo-C’s `CONIO.H`, used to create text user interfaces.
3.5. Compiling Python on Windows

If you want to compile CPython yourself, first thing you should do is get the source. You can download either the latest release’s source or just grab a fresh checkout.

For Microsoft Visual C++, which is the compiler with which official Python releases are built, the source tree contains solutions/project files. View the readme.txt in their respective directories:

<table>
<thead>
<tr>
<th>Directory</th>
<th>MSVC version</th>
<th>Visual Studio version</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC/VC6/</td>
<td>6.0</td>
<td>97</td>
</tr>
<tr>
<td>PC/VS7.1/</td>
<td>7.1</td>
<td>2003</td>
</tr>
<tr>
<td>PC/VS8.0/</td>
<td>8.0</td>
<td>2005</td>
</tr>
<tr>
<td>PCbuild/</td>
<td>9.0</td>
<td>2008</td>
</tr>
</tbody>
</table>

Note that not all of these build directories are fully supported. Read the release notes to see which compiler version the official releases for your version are built with.

Check PC/readme.txt for general information on the build process.

For extension modules, consult Building C and C++ Extensions on Windows.

See also:

Python + Windows + distutils + SWIG + gcc MinGW
or “Creating Python extensions in C/C++ with SWIG and compiling them with MinGW gcc under Windows” or “Installing Python extension with distutils and without Microsoft Visual C++” by Sébastien Sauvage, 2003

MingW – Python extensions
3.6. Other resources

<table>
<thead>
<tr>
<th>See also:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Python Programming On Win32</strong></td>
</tr>
<tr>
<td><strong>A Python for Windows Tutorial</strong></td>
</tr>
<tr>
<td>by Amanda Birmingham, 2004</td>
</tr>
</tbody>
</table>
4. Using Python on a Macintosh

Author: Bob Savage <bobsavage@mac.com>

Python on a Macintosh running Mac OS X is in principle very similar to Python on any other Unix platform, but there are a number of additional features such as the IDE and the Package Manager that are worth pointing out.

The Mac-specific modules are documented in Mac OS X specific services.

Python on Mac OS 9 or earlier can be quite different from Python on Unix or Windows, but is beyond the scope of this manual, as that platform is no longer supported, starting with Python 2.4. See http://www.cwi.nl/~jack/macpython for installers for the latest 2.3 release for Mac OS 9 and related documentation.
4.1. Getting and Installing MacPython

Mac OS X 10.8 comes with Python 2.7 pre-installed by Apple. If you wish, you are invited to install the most recent version of Python from the Python website (https://www.python.org). A current “universal binary” build of Python, which runs natively on the Mac’s new Intel and legacy PPC CPU’s, is available there.

What you get after installing is a number of things:

- A MacPython 2.7 folder in your Applications folder. In here you find IDLE, the development environment that is a standard part of official Python distributions; PythonLauncher, which handles double-clicking Python scripts from the Finder; and the “Build Applet” tool, which allows you to package Python scripts as standalone applications on your system.
- A framework /Library/Frameworks/Python.framework, which includes the Python executable and libraries. The installer adds this location to your shell path. To uninstall MacPython, you can simply remove these three things. A symlink to the Python executable is placed in /usr/local/bin/.

The Apple-provided build of Python is installed in /System/Library/Frameworks/Python.framework and /usr/bin/python, respectively. You should never modify or delete these, as they are Apple-controlled and are used by Apple- or third-party software. Remember that if you choose to install a newer Python version from python.org, you will have two different but functional Python installations on your computer, so it will be important that your paths and usages are consistent with what you want to do.

IDLE includes a help menu that allows you to access Python
documentation. If you are completely new to Python you should start reading the tutorial introduction in that document.

If you are familiar with Python on other Unix platforms you should read the section on running Python scripts from the Unix shell.

### 4.1.1. How to run a Python script

Your best way to get started with Python on Mac OS X is through the IDLE integrated development environment, see section The IDE and use the Help menu when the IDE is running.

If you want to run Python scripts from the Terminal window command line or from the Finder you first need an editor to create your script. Mac OS X comes with a number of standard Unix command line editors, vim and emacs among them. If you want a more Mac-like editor, BBEdit or TextWrangler from Bare Bones Software (see [http://www.barebones.com/products/bbedit/index.html](http://www.barebones.com/products/bbedit/index.html)) are good choices, as is TextMate (see [https://macromates.com/](https://macromates.com/)). Other editors include Gvim ([http://macvim.org](http://macvim.org)) and Aquamacs ([http://aquamacs.org/](http://aquamacs.org/)).

To run your script from the Terminal window you must make sure that /usr/local/bin is in your shell search path.

To run your script from the Finder you have two options:

- Drag it to **PythonLauncher**
- Select **PythonLauncher** as the default application to open your script (or any .py script) through the finder Info window and double-click it. **PythonLauncher** has various preferences to control how your script is launched. Option-dragging allows you to change these for one invocation, or use its Preferences menu to change things globally.
4.1.2. Running scripts with a GUI

With older versions of Python, there is one Mac OS X quirk that you need to be aware of: programs that talk to the Aqua window manager (in other words, anything that has a GUI) need to be run in a special way. Use `pythonw` instead of `python` to start such scripts.

With Python 2.7, you can use either `python` or `pythonw`.

4.1.3. Configuration

Python on OS X honors all standard Unix environment variables such as `PYTHONPATH`, but setting these variables for programs started from the Finder is non-standard as the Finder does not read your `.profile` or `.cshrc` at startup. You need to create a file `~/.MacOSX/environment.plist`. See Apple's Technical Document QA1067 for details.

For more information on installation Python packages in MacPython, see section Installing Additional Python Packages.
4.2. The IDE

MacPython ships with the standard IDLE development environment. A good introduction to using IDLE can be found at https://hkn.eecs.berkeley.edu/~dyoo/python/idle_intro/index.html.
4.3. Installing Additional Python Packages

There are several methods to install additional Python packages:

- Packages can be installed via the standard Python distutils mode (`python setup.py install`).
- Many packages can also be installed via the `setuptools` extension or `pip` wrapper, see https://pip.pypa.io/. 
4.4. GUI Programming on the Mac

There are several options for building GUI applications on the Mac with Python.

*PyObjC* is a Python binding to Apple’s Objective-C/Cocoa framework, which is the foundation of most modern Mac development. Information on PyObjC is available from [https://pythonhosted.org/pyobjc/](https://pythonhosted.org/pyobjc/).

The standard Python GUI toolkit is *Tkinter*, based on the cross-platform Tk toolkit ([https://www.tcl.tk](https://www.tcl.tk)). An Aqua-native version of Tk is bundled with OS X by Apple, and the latest version can be downloaded and installed from [https://www.activestate.com](https://www.activestate.com); it can also be built from source.

*wxPython* is another popular cross-platform GUI toolkit that runs natively on Mac OS X. Packages and documentation are available from [http://www.wxpython.org](http://www.wxpython.org).

*PyQt* is another popular cross-platform GUI toolkit that runs natively on Mac OS X. More information can be found at [https://riverbankcomputing.com/software/pyqt/intro](https://riverbankcomputing.com/software/pyqt/intro).
4.5. Distributing Python Applications on the Mac

The “Build Applet” tool that is placed in the MacPython 2.7 folder is fine for packaging small Python scripts on your own machine to run as a standard Mac application. This tool, however, is not robust enough to distribute Python applications to other users.

The standard tool for deploying standalone Python applications on the Mac is py2app. More information on installing and using py2app can be found at http://undefined.org/python/#py2app.
4.6. Other Resources

The MacPython mailing list is an excellent support resource for Python users and developers on the Mac:

https://www.python.org/community/sigs/current/pythonmac-sig/

Another useful resource is the MacPython wiki:

https://wiki.python.org/moin/MacPython
1. Introduction

This reference manual describes the Python programming language. It is not intended as a tutorial.

While I am trying to be as precise as possible, I chose to use English rather than formal specifications for everything except syntax and lexical analysis. This should make the document more understandable to the average reader, but will leave room for ambiguities. Consequently, if you were coming from Mars and tried to re-implement Python from this document alone, you might have to guess things and in fact you would probably end up implementing quite a different language. On the other hand, if you are using Python and wonder what the precise rules about a particular area of the language are, you should definitely be able to find them here. If you would like to see a more formal definition of the language, maybe you could volunteer your time — or invent a cloning machine :-).

It is dangerous to add too many implementation details to a language reference document — the implementation may change, and other implementations of the same language may work differently. On the other hand, there is currently only one Python implementation in widespread use (although alternate implementations exist), and its particular quirks are sometimes worth being mentioned, especially where the implementation imposes additional limitations. Therefore, you'll find short “implementation notes” sprinkled throughout the text.

Every Python implementation comes with a number of built-in and standard modules. These are documented in The Python Standard Library. A few built-in modules are mentioned when they interact in a significant way with the language definition.
1.1. Alternate Implementations

Though there is one Python implementation which is by far the most popular, there are some alternate implementations which are of particular interest to different audiences.

Known implementations include:

CPython
This is the original and most-maintained implementation of Python, written in C. New language features generally appear here first.

Jython
Python implemented in Java. This implementation can be used as a scripting language for Java applications, or can be used to create applications using the Java class libraries. It is also often used to create tests for Java libraries. More information can be found at the Jython website.

Python for .NET
This implementation actually uses the CPython implementation, but is a managed .NET application and makes .NET libraries available. It was created by Brian Lloyd. For more information, see the Python for .NET home page.

IronPython
An alternate Python for .NET. Unlike Python.NET, this is a complete Python implementation that generates IL, and compiles Python code directly to .NET assemblies. It was created by Jim Hugunin, the original creator of Jython. For more information, see the IronPython website.

PyPy
An implementation of Python written completely in Python. It supports several advanced features not found in other
implementations like stackless support and a Just in Time compiler. One of the goals of the project is to encourage experimentation with the language itself by making it easier to modify the interpreter (since it is written in Python). Additional information is available on the PyPy project's home page.

Each of these implementations varies in some way from the language as documented in this manual, or introduces specific information beyond what's covered in the standard Python documentation. Please refer to the implementation-specific documentation to determine what else you need to know about the specific implementation you're using.
1.2. Notation

The descriptions of lexical analysis and syntax use a modified BNF grammar notation. This uses the following style of definition:

```
name ::= lc_letter (lc_letter | "_")*
lc_letter ::= "a"..."z"
```

The first line says that a name is an lc_letter followed by a sequence of zero or more lc_letters and underscores. An lc_letter in turn is any of the single characters 'a' through 'z'. (This rule is actually adhered to for the names defined in lexical and grammar rules in this document.)

Each rule begins with a name (which is the name defined by the rule) and ::= . A vertical bar (|) is used to separate alternatives; it is the least binding operator in this notation. A star (*) means zero or more repetitions of the preceding item; likewise, a plus (+) means one or more repetitions, and a phrase enclosed in square brackets ([ ]) means zero or one occurrences (in other words, the enclosed phrase is optional). The * and + operators bind as tightly as possible; parentheses are used for grouping. Literal strings are enclosed in quotes. White space is only meaningful to separate tokens. Rules are normally contained on a single line; rules with many alternatives may be formatted alternatively with each line after the first beginning with a vertical bar.

In lexical definitions (as the example above), two more conventions are used: Two literal characters separated by three dots mean a choice of any single character in the given (inclusive) range of ASCII characters. A phrase between angular brackets (<...>) gives an informal description of the symbol defined; e.g., this could be used to describe the notion of ‘control character’ if needed.
Even though the notation used is almost the same, there is a big difference between the meaning of lexical and syntactic definitions: a lexical definition operates on the individual characters of the input source, while a syntax definition operates on the stream of tokens generated by the lexical analysis. All uses of BNF in the next chapter ("Lexical Analysis") are lexical definitions; uses in subsequent chapters are syntactic definitions.
2. Lexical analysis

A Python program is read by a parser. Input to the parser is a stream of tokens, generated by the lexical analyzer. This chapter describes how the lexical analyzer breaks a file into tokens.

Python uses the 7-bit ASCII character set for program text.

*New in version 2.3:* An encoding declaration can be used to indicate that string literals and comments use an encoding different from ASCII.

For compatibility with older versions, Python only warns if it finds 8-bit characters; those warnings should be corrected by either declaring an explicit encoding, or using escape sequences if those bytes are binary data, instead of characters.

The run-time character set depends on the I/O devices connected to the program but is generally a superset of ASCII.

**Future compatibility note:** It may be tempting to assume that the character set for 8-bit characters is ISO Latin-1 (an ASCII superset that covers most western languages that use the Latin alphabet), but it is possible that in the future Unicode text editors will become common. These generally use the UTF-8 encoding, which is also an ASCII superset, but with very different use for the characters with ordinals 128-255. While there is no consensus on this subject yet, it is unwise to assume either Latin-1 or UTF-8, even though the current implementation appears to favor Latin-1. This applies both to the source character set and the run-time character set.
2.1. Line structure

A Python program is divided into a number of logical lines.

2.1.1. Logical lines

The end of a logical line is represented by the token NEWLINE. Statements cannot cross logical line boundaries except where NEWLINE is allowed by the syntax (e.g., between statements in compound statements). A logical line is constructed from one or more physical lines by following the explicit or implicit line joining rules.

2.1.2. Physical lines

A physical line is a sequence of characters terminated by an end-of-line sequence. In source files, any of the standard platform line termination sequences can be used - the Unix form using ASCII LF (linefeed), the Windows form using the ASCII sequence CR LF (return followed by linefeed), or the old Macintosh form using the ASCII CR (return) character. All of these forms can be used equally, regardless of platform.

When embedding Python, source code strings should be passed to Python APIs using the standard C conventions for newline characters (the \n character, representing ASCII LF, is the line terminator).

2.1.3. Comments

A comment starts with a hash character (#) that is not part of a string literal, and ends at the end of the physical line. A comment signifies the end of the logical line unless the implicit line joining rules are
invoked. Comments are ignored by the syntax; they are not tokens.

## 2.1.4. Encoding declarations

If a comment in the first or second line of the Python script matches the regular expression `coding[=:]\s*([-\w.]+)`, this comment is processed as an encoding declaration; the first group of this expression names the encoding of the source code file. The encoding declaration must appear on a line of its own. If it is the second line, the first line must also be a comment-only line. The recommended forms of an encoding expression are

```python
# -*- coding: <encoding-name> -*-
```

which is recognized also by GNU Emacs, and

```python
# vim:fileencoding=<encoding-name>
```

which is recognized by Bram Moolenaar’s VIM. In addition, if the first bytes of the file are the UTF-8 byte-order mark (`\xef\xbb\xbf`), the declared file encoding is UTF-8 (this is supported, among others, by Microsoft’s [notepad](https://docs.microsoft.com/en-us/windows/desktop/winprog Reference#notepad)).

If an encoding is declared, the encoding name must be recognized by Python. The encoding is used for all lexical analysis, in particular to find the end of a string, and to interpret the contents of Unicode literals. String literals are converted to Unicode for syntactical analysis, then converted back to their original encoding before interpretation starts.

## 2.1.5. Explicit line joining

Two or more physical lines may be joined into logical lines using backslash characters (`\`), as follows: when a physical line ends in a
backslash that is not part of a string literal or comment, it is joined with the following forming a single logical line, deleting the backslash and the following end-of-line character. For example:

```
if 1900 < year < 2100 and 1 <= month <= 12 
   and 1 <= day <= 31 and 0 <= hour < 24 
   and 0 <= minute < 60 and 0 <= second < 60:   # Looks like a
   return 1
```

A line ending in a backslash cannot carry a comment. A backslash does not continue a comment. A backslash does not continue a token except for string literals (i.e., tokens other than string literals cannot be split across physical lines using a backslash). A backslash is illegal elsewhere on a line outside a string literal.

### 2.1.6. Implicit line joining

Expressions in parentheses, square brackets or curly braces can be split over more than one physical line without using backslashes. For example:

```
month_names = ['Januari', 'Februari', 'Maart',   # These are
               'April', 'Mei', 'Juni',        # Dutch nam
               'Juli',  'Augustus', 'September', # for the m
               'Oktober', 'November', 'December'] # of the ye
```

Implicitly continued lines can carry comments. The indentation of the continuation lines is not important. Blank continuation lines are allowed. There is no NEWLINE token between implicit continuation lines. Implicitly continued lines can also occur within triple-quoted strings (see below); in that case they cannot carry comments.

### 2.1.7. Blank lines

A logical line that contains only spaces, tabs, formfeeds and possibly
a comment, is ignored (i.e., no NEWLINE token is generated). During interactive input of statements, handling of a blank line may differ depending on the implementation of the read-eval-print loop. In the standard implementation, an entirely blank logical line (i.e. one containing not even whitespace or a comment) terminates a multi-line statement.

### 2.1.8. Indentation

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements.

First, tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to and including the replacement is a multiple of eight (this is intended to be the same rule as used by Unix). The total number of spaces preceding the first non-blank character then determines the line’s indentation. Indentation cannot be split over multiple physical lines using backslashes; the whitespace up to the first backslash determines the indentation.

**Cross-platform compatibility note:** because of the nature of text editors on non-UNIX platforms, it is unwise to use a mixture of spaces and tabs for the indentation in a single source file. It should also be noted that different platforms may explicitly limit the maximum indentation level.

A formfeed character may be present at the start of the line; it will be ignored for the indentation calculations above. Formfeed characters occurring elsewhere in the leading whitespace have an undefined effect (for instance, they may reset the space count to zero).

The indentation levels of consecutive lines are used to generate INDENT and DEDENT tokens, using a stack, as follows.
Before the first line of the file is read, a single zero is pushed on the stack; this will never be popped off again. The numbers pushed on the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, it is pushed on the stack, and one INDENT token is generated. If it is smaller, it must be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a DEDENT token is generated. At the end of the file, a DEDENT token is generated for each number remaining on the stack that is larger than zero.

Here is an example of a correctly (though confusingly) indented piece of Python code:

```python
def perm(l):
    # Compute the list of all permutations of l
    if len(l) <= 1:
        return [l]
    r = []
    for i in range(len(l)):
        s = l[:i] + l[i+1:]
        p = perm(s)
        for x in p:
            r.append(l[i:i+1] + x)
    return r
```

The following example shows various indentation errors:

```python
def perm(l):
    # error: first line indented
    for i in range(len(l)):
        # error: not indented
        s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])
        # error: unexpected indent
        for x in p:
            r.append(l[i:i+1] + x)
    return r
    # error: inconsistent dedent
```

(Actually, the first three errors are detected by the parser; only the
last error is found by the lexical analyzer — the indentation of \texttt{return} does not match a level popped off the stack.)

2.1.9. Whitespace between tokens

Except at the beginning of a logical line or in string literals, the whitespace characters space, tab and formfeed can be used interchangeably to separate tokens. Whitespace is needed between two tokens only if their concatenation could otherwise be interpreted as a different token (e.g., \texttt{ab} is one token, but \texttt{a b} is two tokens).
2.2. Other tokens

Besides NEWLINE, INDENT and DEDENT, the following categories of tokens exist: identifiers, keywords, literals, operators, and delimiters. Whitespace characters (other than line terminators, discussed earlier) are not tokens, but serve to delimit tokens. Where ambiguity exists, a token comprises the longest possible string that forms a legal token, when read from left to right.
2.3. Identifiers and keywords

Identifiers (also referred to as *names*) are described by the following lexical definitions:

```
identifier ::= (letter|"_") (letter | digit | ")*  
letter ::= lowercase | uppercase  
lowercase ::= "a"..."z"  
uppercase ::= "A"..."Z"  
digit ::= "0"..."9"
```

Identifiers are unlimited in length. Case is significant.

2.3.1. Keywords

The following identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

```
and  del  from  not  while
as   elif  global  or  with
assert else  if  pass  yield
break except import print
class exec in  raise
continue finally is return
def  for  lambda  try
```

*Changed in version 2.4:* None became a constant and is now recognized by the compiler as a name for the built-in object None. Although it is not a keyword, you cannot assign a different object to it.

*Changed in version 2.5:* Using as and with as identifiers triggers a warning. To use them as keywords, enable the with_statement future feature.
Changed in version 2.6: `as` and `with` are full keywords.

### 2.3.2. Reserved classes of identifiers

Certain classes of identifiers (besides keywords) have special meanings. These classes are identified by the patterns of leading and trailing underscore characters:

`_*`

Not imported by `from module import *`. The special identifier `_` is used in the interactive interpreter to store the result of the last evaluation; it is stored in the `__builtin__` module. When not in interactive mode, `_` has no special meaning and is not defined. See section *The import statement*.

**Note:** The name `_` is often used in conjunction with internationalization; refer to the documentation for the `gettext` module for more information on this convention.

`___*__`

System-defined names. These names are defined by the interpreter and its implementation (including the standard library). Current system names are discussed in the *Special method names* section and elsewhere. More will likely be defined in future versions of Python. *Any* use of `___*__` names, in any context, that does not follow explicitly documented use, is subject to breakage without warning.

`__*`

Class-private names. Names in this category, when used within the context of a class definition, are re-written to use a mangled form to help avoid name clashes between “private” attributes of base and derived classes. See section *Identifiers (Names)*.
2.4. Literals

Literals are notations for constant values of some built-in types.

2.4.1. String literals

String literals are described by the following lexical definitions:

\[
\begin{align*}
\text{stringliteral} & ::= \ [\text{stringprefix}] (\text{shortstring} \mid \text{longstring}) \\
\text{stringprefix} & ::= \ "r" \mid \ "u" \mid \ "ur" \mid \ "R" \mid \ "U" \mid \ "UR" \mid \ "Ur" \\
& \mid \ "b" \mid \ "B" \mid \ "br" \mid \ "Br" \mid \ "bR" \mid \ "BR" \\
\text{shortstring} & ::= \ ''' \text{shortstringitem}''' \mid \ ''' \\
\text{longstring} & ::= \ ''''' \text{longstringitem}'''''' \mid \ ''''''' \\
\text{shortstringitem} & ::= \ \text{shortstringchar} \mid \ \text{escapeseq} \\
\text{longstringitem} & ::= \ \text{longstringchar} \mid \ \text{escapeseq} \\
\text{shortstringchar} & ::= \ \langle \text{any source character except } "\" \text{ or newline} \rangle \\
\text{longstringchar} & ::= \ \langle \text{any source character except } "\" \rangle \\
\text{escapeseq} & ::= \ \"\" \langle \text{any ASCII character} \rangle
\end{align*}
\]

One syntactic restriction not indicated by these productions is that whitespace is not allowed between the \text{stringprefix} and the rest of the string literal. The source character set is defined by the encoding declaration; it is ASCII if no encoding declaration is given in the source file; see section Encoding declarations.

In plain English: String literals can be enclosed in matching single quotes (') or double quotes ("). They can also be enclosed in matching groups of three single or double quotes (these are generally referred to as \textit{triple-quoted strings}). The backslash (\) character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character. String literals may optionally be prefixed with a letter 'r' or 'R'; such strings are called \textit{raw strings} and use different rules for interpreting backslash escape sequences. A prefix of 'u' or 'U'
makes the string a Unicode string. Unicode strings use the Unicode character set as defined by the Unicode Consortium and ISO 10646. Some additional escape sequences, described below, are available in Unicode strings. A prefix of \texttt{b'} or \texttt{B'} is ignored in Python 2; it indicates that the literal should become a bytes literal in Python 3 (e.g. when code is automatically converted with 2to3). A \texttt{u'} or \texttt{b'} prefix may be followed by an \texttt{r'} prefix.

In triple-quoted strings, unescaped newlines and quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the string. (A “quote” is the character used to open the string, i.e. either \texttt{'} or \texttt{"}).

Unless an \texttt{r'} or \texttt{R'} prefix is present, escape sequences in strings are interpreted according to rules similar to those used by Standard C. The recognized escape sequences are:

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>\newline</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td>\</td>
<td>Backslash ()</td>
<td></td>
</tr>
<tr>
<td>'</td>
<td>Single quote (')</td>
<td></td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote (&quot;)</td>
<td></td>
</tr>
<tr>
<td>\a</td>
<td>ASCII Bell (BEL)</td>
<td></td>
</tr>
<tr>
<td>\b</td>
<td>ASCII Backspace (BS)</td>
<td></td>
</tr>
<tr>
<td>\f</td>
<td>ASCII Formfeed (FF)</td>
<td></td>
</tr>
<tr>
<td>\n</td>
<td>ASCII Linefeed (LF)</td>
<td></td>
</tr>
<tr>
<td>\N{name}</td>
<td>Character named name in the Unicode database (Unicode only)</td>
<td></td>
</tr>
<tr>
<td>\r</td>
<td>ASCII Carriage Return (CR)</td>
<td></td>
</tr>
<tr>
<td>\t</td>
<td>ASCII Horizontal Tab (TAB)</td>
<td></td>
</tr>
<tr>
<td>\uxxxx</td>
<td>Character with 16-bit hex value xxxx (Unicode only)</td>
<td>(1)</td>
</tr>
<tr>
<td>\Uxxxxxxxxxxx</td>
<td>Character with 32-bit hex value xxxxxxxx (Unicode only)</td>
<td>(2)</td>
</tr>
<tr>
<td>Backslash</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>\v</td>
<td>ASCII Vertical Tab (VT)</td>
<td></td>
</tr>
<tr>
<td>\ooo</td>
<td>Character with octal value ooo (3,5)</td>
<td></td>
</tr>
<tr>
<td>\xhh</td>
<td>Character with hex value hh (4,5)</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. Individual code units which form parts of a surrogate pair can be encoded using this escape sequence.
2. Any Unicode character can be encoded this way, but characters outside the Basic Multilingual Plane (BMP) will be encoded using a surrogate pair if Python is compiled to use 16-bit code units (the default).
3. As in Standard C, up to three octal digits are accepted.
4. Unlike in Standard C, exactly two hex digits are required.
5. In a string literal, hexadecimal and octal escapes denote the byte with the given value; it is not necessary that the byte encodes a character in the source character set. In a Unicode literal, these escapes denote a Unicode character with the given value.

Unlike Standard C, all unrecognized escape sequences are left in the string unchanged, i.e., *the backslash is left in the string*. (This behavior is useful when debugging: if an escape sequence is mistyped, the resulting output is more easily recognized as broken.) It is also important to note that the escape sequences marked as “(Unicode only)” in the table above fall into the category of unrecognized escapes for non-Unicode string literals.

When an `\r` or `\R` prefix is present, a character following a backslash is included in the string without change, and *all backslashes are left in the string*. For example, the string literal `r"\n"` consists of two characters: a backslash and a lowercase `\n`. String quotes can be escaped with a backslash, but the backslash remains in the string; for example, `r"\""` is a valid string literal consisting of
two characters: a backslash and a double quote; `r"\"` is not a valid string literal (even a raw string cannot end in an odd number of backslashes). Specifically, a raw string cannot end in a single backslash (since the backslash would escape the following quote character). Note also that a single backslash followed by a newline is interpreted as those two characters as part of the string, not as a line continuation.

When an `r` or `R` prefix is used in conjunction with a `u` or `U` prefix, then the \uxxxx and \uxxxxxx escape sequences are processed while all other backslashes are left in the string. For example, the string literal ur"\u0062\n" consists of three Unicode characters: ‘LATIN SMALL LETTER B’, ‘REVERSE SOLIDUS’, and ‘LATIN SMALL LETTER N’. Backslashes can be escaped with a preceding backslash; however, both remain in the string. As a result, \uxxxx escape sequences are only recognized when there are an odd number of backslashes.

### 2.4.2. String literal concatenation

Multiple adjacent string literals (delimited by whitespace), possibly using different quoting conventions, are allowed, and their meaning is the same as their concatenation. Thus, "hello" 'world' is equivalent to "helloworld". This feature can be used to reduce the number of backslashes needed, to split long strings conveniently across long lines, or even to add comments to parts of strings, for example:

```python
re.compile("[A-Za-z_]" # letter or underscore
           "[A-Za-z0-9_]"* # letter, digit or underscore
)
```

Note that this feature is defined at the syntactical level, but implemented at compile time. The ‘+’ operator must be used to
concatenate string expressions at run time. Also note that literal concatenation can use different quoting styles for each component (even mixing raw strings and triple quoted strings).

### 2.4.3. Numeric literals

There are four types of numeric literals: plain integers, long integers, floating point numbers, and imaginary numbers. There are no complex literals (complex numbers can be formed by adding a real number and an imaginary number).

Note that numeric literals do not include a sign; a phrase like -1 is actually an expression composed of the unary operator ‘-’ and the literal 1.

### 2.4.4. Integer and long integer literals

Integer and long integer literals are described by the following lexical definitions:

```
longinteger ::= integer ("l" | "L")
integer ::= decimalinteger | octinteger | hexinteger | longinteger
decimalinteger ::= nonzerodigit digit* | "0"
octinteger ::= "0" ("o" | "O") octdigit+ | "0" octdigit+
hexinteger ::= "0" ("x" | "X") hexdigit+
bininteger ::= "0" ("b" | "B") bindigit+
nonzerodigit ::= "1"..."9"
octdigit ::= "0"..."7"
bindigit ::= "0" | "1"
hexdigit ::= digit | "a"..."f" | "A"..."F"
```

Although both lower case 'l' and upper case 'L' are allowed as suffix for long integers, it is strongly recommended to always use 'l', since the letter 'l' looks too much like the digit '1'.

Plain integer literals that are above the largest representable plain
integer (e.g., 2147483647 when using 32-bit arithmetic) are accepted as if they were long integers instead. [1] There is no limit for long integer literals apart from what can be stored in available memory.

Some examples of plain integer literals (first row) and long integer literals (second and third rows):

<table>
<thead>
<tr>
<th></th>
<th>Integer Literals</th>
<th>Long Integer Literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2147483647</td>
<td>0177</td>
</tr>
<tr>
<td>3L</td>
<td>7922816251426437593543950336L</td>
<td>0377L</td>
</tr>
<tr>
<td></td>
<td>7922816251426437593543950336</td>
<td>0xdeadbeef</td>
</tr>
</tbody>
</table>

2.4.5. Floating point literals

Floating point literals are described by the following lexical definitions:

\[
\begin{align*}
\text{floatnumber} & ::= \text{pointfloat} \mid \text{exponentfloat} \\
\text{pointfloat} & ::= [\text{intpart}] \ \text{fraction} \mid \text{intpart } \cdot \\
\text{exponentfloat} & ::= (\text{intpart} \mid \text{pointfloat}) \ \text{exponent} \\
\text{intpart} & ::= \text{digit} + \\
\text{fraction} & ::= \cdot \text{digit} + \\
\text{exponent} & ::= (\text{e} \mid \text{E}) [^+ \mid -] \text{digit} +
\end{align*}
\]

Note that the integer and exponent parts of floating point numbers can look like octal integers, but are interpreted using radix 10. For example, \(077e010\) is legal, and denotes the same number as \(77e10\). The allowed range of floating point literals is implementation-dependent. Some examples of floating point literals:

|    | 3.14 | 10.  | .001 | 1e100 | 3.14e-10 | 0e0  |

Note that numeric literals do not include a sign; a phrase like \(-1\) is actually an expression composed of the unary operator \(-\) and the literal \(1\).
2.4.6. Imaginary literals

Imaginary literals are described by the following lexical definitions:

```
imagnumber ::= (floatnumber | intpart) ("j" | "J")
```

An imaginary literal yields a complex number with a real part of 0.0. Complex numbers are represented as a pair of floating point numbers and have the same restrictions on their range. To create a complex number with a nonzero real part, add a floating point number to it, e.g., (3+4j). Some examples of imaginary literals:

```
3.14j  10.j  10j  .001j  1e100j  3.14e-10j
```
2.5. Operators

The following tokens are operators:

| + | - | * | ** | / | // | % |
| << | >> | & | | ^ | ~ |
< | > | <= | >= | == | != | <> |

The comparison operators <> and != are alternate spellings of the same operator. != is the preferred spelling; <> is obsolescent.
2.6. Delimiters

The following tokens serve as delimiters in the grammar:

```
(      )     [      ]     {      }     @
,      :      .      `      =      ;
+=     -=     *=     /=     //=     %=  
&=     |=     ^=     >>=     <<=     **=
```

The period can also occur in floating-point and imaginary literals. A sequence of three periods has a special meaning as an ellipsis in slices. The second half of the list, the augmented assignment operators, serve lexically as delimiters, but also perform an operation.

The following printing ASCII characters have special meaning as part of other tokens or are otherwise significant to the lexical analyzer:

```
'   "   #   \```

The following printing ASCII characters are not used in Python. Their occurrence outside string literals and comments is an unconditional error:

```
$   ?```

Footnotes

[1] In versions of Python prior to 2.4, octal and hexadecimal literals in the range just above the largest representable plain integer but below the largest unsigned 32-bit number (on a machine using 32-bit arithmetic), 4294967296, were taken as the negative plain integer obtained by subtracting 4294967296 from their unsigned value.
3. Data model
3.1. Objects, values and types

Objects are Python’s abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann’s model of a “stored program computer,” code is also represented by objects.)

Every object has an identity, a type and a value. An object’s identity never changes once it has been created; you may think of it as the object’s address in memory. The ‘is’ operator compares the identity of two objects; the \texttt{id()} function returns an integer representing its identity (currently implemented as its address). An object’s type is also unchangeable. [1] An object’s type determines the operations that the object supports (e.g., “does it have a length?”) and also defines the possible values for objects of that type. The \texttt{type()} function returns an object’s type (which is an object itself). The value of some objects can change. Objects whose value can change are said to be \textit{mutable}; objects whose value is unchangeable once they are created are called \textit{immutable}. (The value of an immutable container object that contains a reference to a mutable object can change when the latter’s value is changed; however the container is still considered immutable, because the collection of objects it contains cannot be changed. So, immutability is not strictly the same as having an unchangeable value, it is more subtle.) An object’s mutability is determined by its type; for instance, numbers, strings and tuples are immutable, while dictionaries and lists are mutable.

Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected. An implementation is allowed to postpone garbage collection or omit it altogether — it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable.
**CPython implementation detail:** CPython currently uses a reference-counting scheme with (optional) delayed detection of cyclically linked garbage, which collects most objects as soon as they become unreachable, but is not guaranteed to collect garbage containing circular references. See the documentation of the `gc` module for information on controlling the collection of cyclic garbage. Other implementations act differently and CPython may change. Do not depend on immediate finalization of objects when they become unreachable (ex: always close files).

Note that the use of the implementation’s tracing or debugging facilities may keep objects alive that would normally be collectable. Also note that catching an exception with a `try...except` statement may keep objects alive.

Some objects contain references to “external” resources such as open files or windows. It is understood that these resources are freed when the object is garbage-collected, but since garbage collection is not guaranteed to happen, such objects also provide an explicit way to release the external resource, usually a `close()` method. Programs are strongly recommended to explicitly close such objects. The `try...finally` statement provides a convenient way to do this.

Some objects contain references to other objects; these are called *containers*. Examples of containers are tuples, lists and dictionaries. The references are part of a container’s value. In most cases, when we talk about the value of a container, we imply the values, not the identities of the contained objects; however, when we talk about the mutability of a container, only the identities of the immediately contained objects are implied. So, if an immutable container (like a tuple) contains a reference to a mutable object, its value changes if
that mutable object is changed.

Types affect almost all aspects of object behavior. Even the importance of object identity is affected in some sense: for immutable types, operations that compute new values may actually return a reference to any existing object with the same type and value, while for mutable objects this is not allowed. E.g., after `a = 1; b = 1`, `a` and `b` may or may not refer to the same object with the value one, depending on the implementation, but after `c = []; d = []`, `c` and `d` are guaranteed to refer to two different, unique, newly created empty lists. (Note that `c = d = []` assigns the same object to both `c` and `d`.)
3.2. The standard type hierarchy

Below is a list of the types that are built into Python. Extension modules (written in C, Java, or other languages, depending on the implementation) can define additional types. Future versions of Python may add types to the type hierarchy (e.g., rational numbers, efficiently stored arrays of integers, etc.).

Some of the type descriptions below contain a paragraph listing ‘special attributes.’ These are attributes that provide access to the implementation and are not intended for general use. Their definition may change in the future.

None

This type has a single value. There is a single object with this value. This object is accessed through the built-in name `None`. It is used to signify the absence of a value in many situations, e.g., it is returned from functions that don’t explicitly return anything. Its truth value is false.

NotImplemented

This type has a single value. There is a single object with this value. This object is accessed through the built-in name `NotImplemented`. Numeric methods and rich comparison methods may return this value if they do not implement the operation for the operands provided. (The interpreter will then try the reflected operation, or some other fallback, depending on the operator.) Its truth value is true.

Ellipsis

This type has a single value. There is a single object with this value. This object is accessed through the built-in name `Ellipsis`. It is used to indicate the presence of the `...` syntax in
a slice. Its truth value is true.

**numbers.Number**

These are created by numeric literals and returned as results by arithmetic operators and arithmetic built-in functions. Numeric objects are immutable; once created their value never changes. Python numbers are of course strongly related to mathematical numbers, but subject to the limitations of numerical representation in computers.

Python distinguishes between integers, floating point numbers, and complex numbers:

**numbers.Integral**

These represent elements from the mathematical set of integers (positive and negative).

There are three types of integers:

**Plain integers**

These represent numbers in the range -2147483648 through 2147483647. (The range may be larger on machines with a larger natural word size, but not smaller.) When the result of an operation would fall outside this range, the result is normally returned as a long integer (in some cases, the exception `OverflowError` is raised instead). For the purpose of shift and mask operations, integers are assumed to have a binary, 2’s complement notation using 32 or more bits, and hiding no bits from the user (i.e., all 4294967296 different bit patterns correspond to different values).

**Long integers**

These represent numbers in an unlimited range, subject to available (virtual) memory only. For the purpose of shift
and mask operations, a binary representation is assumed, and negative numbers are represented in a variant of 2’s complement which gives the illusion of an infinite string of sign bits extending to the left.

Booleans

These represent the truth values False and True. The two objects representing the values False and True are the only Boolean objects. The Boolean type is a subtype of plain integers, and Boolean values behave like the values 0 and 1, respectively, in almost all contexts, the exception being that when converted to a string, the strings "False" or "True" are returned, respectively.

The rules for integer representation are intended to give the most meaningful interpretation of shift and mask operations involving negative integers and the least surprises when switching between the plain and long integer domains. Any operation, if it yields a result in the plain integer domain, will yield the same result in the long integer domain or when using mixed operands. The switch between domains is transparent to the programmer.

**numbers.Real (float)**

These represent machine-level double precision floating point numbers. You are at the mercy of the underlying machine architecture (and C or Java implementation) for the accepted range and handling of overflow. Python does not support single-precision floating point numbers; the savings in processor and memory usage that are usually the reason for using these are dwarfed by the overhead of using objects in Python, so there is no reason to complicate the language with two kinds of floating point numbers.
numbers.Complex

These represent complex numbers as a pair of machine-level double precision floating point numbers. The same caveats apply as for floating point numbers. The real and imaginary parts of a complex number \( z \) can be retrieved through the read-only attributes \( z\text{.real} \) and \( z\text{.imag} \).

Sequences

These represent finite ordered sets indexed by non-negative numbers. The built-in function \( \text{len()} \) returns the number of items of a sequence. When the length of a sequence is \( n \), the index set contains the numbers 0, 1, ..., \( n-1 \). Item \( i \) of sequence \( a \) is selected by \( a[i] \).

Sequences also support slicing: \( a[i:j] \) selects all items with index \( k \) such that \( i \leq k < j \). When used as an expression, a slice is a sequence of the same type. This implies that the index set is renumbered so that it starts at 0.

Some sequences also support “extended slicing” with a third “step” parameter: \( a[i:j:k] \) selects all items of \( a \) with index \( x \) where \( x = i + n\times k \), \( n \geq 0 \) and \( i \leq x < j \).

Sequences are distinguished according to their mutability:

Immutable sequences

An object of an immutable sequence type cannot change once it is created. (If the object contains references to other objects, these other objects may be mutable and may be changed; however, the collection of objects directly referenced by an immutable object cannot change.)

The following types are immutable sequences:

Strings
The items of a string are characters. There is no separate character type; a character is represented by a string of one item. Characters represent (at least) 8-bit bytes. The built-in functions `chr()` and `ord()` convert between characters and nonnegative integers representing the byte values. Bytes with the values 0–127 usually represent the corresponding ASCII values, but the interpretation of values is up to the program. The string data type is also used to represent arrays of bytes, e.g., to hold data read from a file.

(On systems whose native character set is not ASCII, strings may use EBCDIC in their internal representation, provided the functions `chr()` and `ord()` implement a mapping between ASCII and EBCDIC, and string comparison preserves the ASCII order. Or perhaps someone can propose a better rule?)

Unicode

The items of a Unicode object are Unicode code units. A Unicode code unit is represented by a Unicode object of one item and can hold either a 16-bit or 32-bit value representing a Unicode ordinal (the maximum value for the ordinal is given in `sys.maxunicode`, and depends on how Python is configured at compile time). Surrogate pairs may be present in the Unicode object, and will be reported as two separate items. The built-in functions `unichr()` and `ord()` convert between code units and nonnegative integers representing the Unicode ordinals as defined in the Unicode Standard 3.0. Conversion from and to other encodings are possible through the Unicode method `encode()` and the built-in function `unicode()`.

Tuples
The items of a tuple are arbitrary Python objects. Tuples of two or more items are formed by comma-separated lists of expressions. A tuple of one item (a ‘singleton’) can be formed by affixing a comma to an expression (an expression by itself does not create a tuple, since parentheses must be usable for grouping of expressions). An empty tuple can be formed by an empty pair of parentheses.

Mutable sequences

Mutable sequences can be changed after they are created. The subscription and slicing notations can be used as the target of assignment and del (delete) statements.

There are currently two intrinsic mutable sequence types:

Lists

The items of a list are arbitrary Python objects. Lists are formed by placing a comma-separated list of expressions in square brackets. (Note that there are no special cases needed to form lists of length 0 or 1.)

Byte Arrays

A bytearray object is a mutable array. They are created by the built-in bytearray() constructor. Aside from being mutable (and hence unhashable), byte arrays otherwise provide the same interface and functionality as immutable bytes objects.

The extension module array provides an additional example of a mutable sequence type.

Set types

These represent unordered, finite sets of unique, immutable objects. As such, they cannot be indexed by any subscript.
However, they can be iterated over, and the built-in function \texttt{len()} returns the number of items in a set. Common uses for sets are fast membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.

For set elements, the same immutability rules apply as for dictionary keys. Note that numeric types obey the normal rules for numeric comparison: if two numbers compare equal (e.g., 1 and 1.0), only one of them can be contained in a set.

There are currently two intrinsic set types:

Sets
These represent a mutable set. They are created by the built-in \texttt{set()} constructor and can be modified afterwards by several methods, such as \texttt{add()}.

Frozen sets
These represent an immutable set. They are created by the built-in \texttt{frozenset()} constructor. As a frozenset is immutable and hashable, it can be used again as an element of another set, or as a dictionary key.

Mappings
These represent finite sets of objects indexed by arbitrary index sets. The subscript notation \texttt{a[k]} selects the item indexed by \texttt{k} from the mapping \texttt{a}; this can be used in expressions and as the target of assignments or \texttt{del} statements. The built-in function \texttt{len()} returns the number of items in a mapping.

There is currently a single intrinsic mapping type:

Dictionaries
These represent finite sets of objects indexed by nearly arbitrary values. The only types of values not acceptable as keys are values containing lists or dictionaries or other mutable types that are compared by value rather than by object identity, the reason being that the efficient implementation of dictionaries requires a key's hash value to remain constant. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (e.g., 1 and 1.0) then they can be used interchangeably to index the same dictionary entry.

Dictionaries are mutable; they can be created by the `{...}` notation (see section Dictionary displays).

The extension modules `dbm`, `gdbm`, and `bsddb` provide additional examples of mapping types.

Callable types

These are the types to which the function call operation (see section Calls) can be applied:

User-defined functions

A user-defined function object is created by a function definition (see section Function definitions). It should be called with an argument list containing the same number of items as the function's formal parameter list.

Special attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__doc__</code> func_doc</td>
<td>The function's documentation string, or None if unavailable.</td>
</tr>
<tr>
<td><code>__name__</code> func_name</td>
<td>The function's name</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>module</strong></td>
<td>function was defined in, or None if unavailable.</td>
</tr>
<tr>
<td><strong>defaults</strong></td>
<td>A tuple containing default argument values for those arguments that have defaults, or None if no arguments have a default value.</td>
</tr>
<tr>
<td>func_defaults</td>
<td></td>
</tr>
<tr>
<td><strong>code</strong></td>
<td>The code object representing the compiled function body.</td>
</tr>
<tr>
<td>func_code</td>
<td></td>
</tr>
<tr>
<td><strong>globals</strong></td>
<td>A reference to the dictionary that holds the function’s global variables — the global namespace of the module in which the function was defined.</td>
</tr>
<tr>
<td>func_globals</td>
<td></td>
</tr>
<tr>
<td><strong>dict</strong></td>
<td>The namespace supporting arbitrary function attributes.</td>
</tr>
<tr>
<td>func_dict</td>
<td></td>
</tr>
<tr>
<td><strong>closure</strong></td>
<td>None or a tuple of cells that contain bindings for the function’s free variables.</td>
</tr>
<tr>
<td>func_closure</td>
<td></td>
</tr>
</tbody>
</table>

Most of the attributes labelled “Writable” check the type of the assigned value.

_Changed in version 2.4:_ func_name is now writable.

_Changed in version 2.6:_ The double-underscore attributes __closure__, __code__, __defaults__, and __globals__ were introduced as aliases for the corresponding func_* attributes for forwards compatibility with Python 3.

Function objects also support getting and setting arbitrary attributes, which can be used, for example, to attach metadata to functions. Regular attribute dot-notation is used to get and set such attributes. _Note that the current_
Implementation only supports function attributes on user-defined functions. Function attributes on built-in functions may be supported in the future.

Additional information about a function's definition can be retrieved from its code object; see the description of internal types below.

User-defined methods
A user-defined method object combines a class, a class instance (or None) and any callable object (normally a user-defined function).

Special read-only attributes: im_self is the class instance object, im_func is the function object; im_class is the class of im_self for bound methods or the class that asked for the method for unbound methods; __doc__ is the method's documentation (same as im_func.__doc__); __name__ is the method name (same as im_func.__name__); __module__ is the name of the module the method was defined in, or None if unavailable.

Changed in version 2.2: im_self used to refer to the class that defined the method.

Changed in version 2.6: For Python 3 forward-compatibility, im_func is also available as __func__, and im_self as __self__.

Methods also support accessing (but not setting) the arbitrary function attributes on the underlying function object.

User-defined method objects may be created when getting an attribute of a class (perhaps via an instance of that class), if
that attribute is a user-defined function object, an unbound user-defined method object, or a class method object. When the attribute is a user-defined method object, a new method object is only created if the class from which it is being retrieved is the same as, or a derived class of, the class stored in the original method object; otherwise, the original method object is used as it is.

When a user-defined method object is created by retrieving a user-defined function object from a class, its `im_self` attribute is `None` and the method object is said to be unbound. When one is created by retrieving a user-defined function object from a class via one of its instances, its `im_self` attribute is the instance, and the method object is said to be bound. In either case, the new method’s `im_class` attribute is the class from which the retrieval takes place, and its `im_func` attribute is the original function object.

When a user-defined method object is created by retrieving another method object from a class or instance, the behaviour is the same as for a function object, except that the `im_func` attribute of the new instance is not the original method object but its `im_func` attribute.

When a user-defined method object is created by retrieving a class method object from a class or instance, its `im_self` attribute is the class itself, and its `im_func` attribute is the function object underlying the class method.

When an unbound user-defined method object is called, the underlying function (`im_func`) is called, with the restriction that the first argument must be an instance of the proper class (`im_class`) or of a derived class thereof.
When a bound user-defined method object is called, the underlying function (\texttt{im\_func}) is called, inserting the class instance (\texttt{im\_self}) in front of the argument list. For instance, when \texttt{c} is a class which contains a definition for a function \texttt{f()}, and \texttt{x} is an instance of \texttt{c}, calling \texttt{x.f(1)} is equivalent to calling \texttt{c.f(x, 1)}.

When a user-defined method object is derived from a class method object, the “class instance” stored in \texttt{im\_self} will actually be the class itself, so that calling either \texttt{x.f(1)} or \texttt{c.f(1)} is equivalent to calling \texttt{f(c, 1)} where \texttt{f} is the underlying function.

Note that the transformation from function object to (unbound or bound) method object happens each time the attribute is retrieved from the class or instance. In some cases, a fruitful optimization is to assign the attribute to a local variable and call that local variable. Also notice that this transformation only happens for user-defined functions; other callable objects (and all non-callable objects) are retrieved without transformation. It is also important to note that user-defined functions which are attributes of a class instance are not converted to bound methods; this \emph{only} happens when the function is an attribute of the class.

\textbf{Generator functions}

A function or method which uses the \texttt{yield} statement (see section \textit{The yield statement}) is called a \textit{generator function}. Such a function, when called, always returns an iterator object which can be used to execute the body of the function: calling the iterator’s \texttt{next()} method will cause the function to execute until it provides a value using the \texttt{yield} statement. When the function executes a \texttt{return} statement or falls off the end, a
StopIteration exception is raised and the iterator will have reached the end of the set of values to be returned.

Built-in functions
A built-in function object is a wrapper around a C function. Examples of built-in functions are `len()` and `math.sin()` (math is a standard built-in module). The number and type of the arguments are determined by the C function. Special read-only attributes: `__doc__` is the function's documentation string, or `None` if unavailable; `__name__` is the function's name; `__self__` is set to `None` (but see the next item); `__module__` is the name of the module the function was defined in or `None` if unavailable.

Built-in methods
This is really a different disguise of a built-in function, this time containing an object passed to the C function as an implicit extra argument. An example of a built-in method is `alist.append()`, assuming `alist` is a list object. In this case, the special read-only attribute `__self__` is set to the object denoted by `alist`.

Class Types
Class types, or “new-style classes,” are callable. These objects normally act as factories for new instances of themselves, but variations are possible for class types that override `__new__()`. The arguments of the call are passed to `__new__()` and, in the typical case, to `__init__()` to initialize the new instance.

Classic Classes
Class objects are described below. When a class object is called, a new class instance (also described below) is created and returned. This implies a call to the class's `__init__()`
method if it has one. Any arguments are passed on to the __init__() method. If there is no __init__() method, the class must be called without arguments.

Class instances

Class instances are described below. Class instances are callable only when the class has a __call__() method; x(arguments) is a shorthand for x.__call__(arguments).

Modules

Modules are imported by the import statement (see section The import statement). A module object has a namespace implemented by a dictionary object (this is the dictionary referenced by the func_globals attribute of functions defined in the module). Attribute references are translated to lookups in this dictionary, e.g., m.x is equivalent to m.__dict__["x"]. A module object does not contain the code object used to initialize the module (since it isn’t needed once the initialization is done).

Attribute assignment updates the module’s namespace dictionary, e.g., m.x = 1 is equivalent to m.__dict__["x"] = 1.

Special read-only attribute: __dict__ is the module’s namespace as a dictionary object.

**CPython implementation detail:** Because of the way CPython clears module dictionaries, the module dictionary will be cleared when the module falls out of scope even if the dictionary still has live references. To avoid this, copy the dictionary or keep the module around while using its dictionary directly.

Predefined (writable) attributes: __name__ is the module’s name; __doc__ is the module’s documentation string, or None if
unavailable; __file__ is the pathname of the file from which the module was loaded, if it was loaded from a file. The __file__ attribute is not present for C modules that are statically linked into the interpreter; for extension modules loaded dynamically from a shared library, it is the pathname of the shared library file.

Classes
Both class types (new-style classes) and class objects (old-style/classic classes) are typically created by class definitions (see section Class definitions). A class has a namespace implemented by a dictionary object. Class attribute references are translated to lookups in this dictionary, e.g., c.x is translated to c.__dict__["x"] (although for new-style classes in particular there are a number of hooks which allow for other means of locating attributes). When the attribute name is not found there, the attribute search continues in the base classes. For old-style classes, the search is depth-first, left-to-right in the order of occurrence in the base class list. New-style classes use the more complex C3 method resolution order which behaves correctly even in the presence of ‘diamond’ inheritance structures where there are multiple inheritance paths leading back to a common ancestor. Additional details on the C3 MRO used by new-style classes can be found in the documentation accompanying the 2.3 release at https://www.python.org/download/releases/2.3/mro/.

When a class attribute reference (for class c, say) would yield a user-defined function object or an unbound user-defined method object whose associated class is either c or one of its base classes, it is transformed into an unbound user-defined method object whose im_class attribute is c. When it would yield a class method object, it is transformed into a bound user-defined method object whose im_self attribute is c. When it would yield a static method object, it is transformed into the object wrapped by the static method object. See section Implementing Descriptors
for another way in which attributes retrieved from a class may differ from those actually contained in its \_dict\_ (note that only new-style classes support descriptors).

Class attribute assignments update the class's dictionary, never the dictionary of a base class.

A class object can be called (see above) to yield a class instance (see below).

Special attributes: \_name\_ is the class name; \_module\_ is the module name in which the class was defined; \_dict\_ is the dictionary containing the class’s namespace; \_bases\_ is a tuple (possibly empty or a singleton) containing the base classes, in the order of their occurrence in the base class list; \_doc\_ is the class’s documentation string, or None if undefined.

Class instances
A class instance is created by calling a class object (see above). A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched. When an attribute is not found there, and the instance's class has an attribute by that name, the search continues with the class attributes. If a class attribute is found that is a user-defined function object or an unbound user-defined method object whose associated class is the class (call it \textit{c}) of the instance for which the attribute reference was initiated or one of its bases, it is transformed into a bound user-defined method object whose \texttt{im\_class} attribute is \textit{c} and whose \texttt{im\_self} attribute is the instance. Static method and class method objects are also transformed, as if they had been retrieved from class \textit{c}; see above under “Classes”. See section \textbf{Implementing Descriptors} for another way in which attributes of a class retrieved via its instances may differ from the objects actually stored in the class's
If no class attribute is found, and the object's class has a `__getattr__()` method, that is called to satisfy the lookup.

Attribute assignments and deletions update the instance's dictionary, never a class's dictionary. If the class has a `__setattr__()` or `__delattr__()` method, this is called instead of updating the instance dictionary directly.

Class instances can pretend to be numbers, sequences, or mappings if they have methods with certain special names. See section Special method names.

Special attributes: `__dict__` is the attribute dictionary; `__class__` is the instance's class.

Files
A file object represents an open file. File objects are created by the `open()` built-in function, and also by `os.popen()`, `os.fdopen()`, and the `makefile()` method of socket objects (and perhaps by other functions or methods provided by extension modules). The objects `sys.stdin`, `sys.stdout` and `sys.stderr` are initialized to file objects corresponding to the interpreter's standard input, output and error streams. See File Objects for complete documentation of file objects.

Internal types
A few types used internally by the interpreter are exposed to the user. Their definitions may change with future versions of the interpreter, but they are mentioned here for completeness.

Code objects
Code objects represent byte-compiled executable Python code, or bytecode. The difference between a code object and a function object is that the function object contains an explicit reference to the function’s globals (the module in which it was
defined), while a code object contains no context; also the default argument values are stored in the function object, not in the code object (because they represent values calculated at run-time). Unlike function objects, code objects are immutable and contain no references (directly or indirectly) to mutable objects.

Special read-only attributes: co_name gives the function name; co_argcount is the number of positional arguments (including arguments with default values); co_nlocals is the number of local variables used by the function (including arguments); co_varnames is a tuple containing the names of the local variables (starting with the argument names); co_cellvars is a tuple containing the names of local variables that are referenced by nested functions; co_freevars is a tuple containing the names of free variables; co_code is a string representing the sequence of bytecode instructions; co_consts is a tuple containing the literals used by the bytecode; co_names is a tuple containing the names used by the bytecode; co_filename is the filename from which the code was compiled; co_firstlineno is the first line number of the function; co_lnotab is a string encoding the mapping from bytecode offsets to line numbers (for details see the source code of the interpreter); co_stacksize is the required stack size (including local variables); co_flags is an integer encoding a number of flags for the interpreter.

The following flag bits are defined for co_flags: bit 0x04 is set if the function uses the *arguments syntax to accept an arbitrary number of positional arguments; bit 0x08 is set if the function uses the **keywords syntax to accept arbitrary keyword arguments; bit 0x20 is set if the function is a
generator.

Future feature declarations (from __future__ import division) also use bits in co_flags to indicate whether a code object was compiled with a particular feature enabled: bit 0x2000 is set if the function was compiled with future division enabled; bits 0x10 and 0x1000 were used in earlier versions of Python.

Other bits in co_flags are reserved for internal use.

If a code object represents a function, the first item in co_consts is the documentation string of the function, or None if undefined.

Frame objects
Frame objects represent execution frames. They may occur in traceback objects (see below).

Special read-only attributes: f_back is to the previous stack frame (towards the caller), or None if this is the bottom stack frame; f_code is the code object being executed in this frame; f_locals is the dictionary used to look up local variables; f_globals is used for global variables; f_builtins is used for built-in (intrinsic) names; f_restricted is a flag indicating whether the function is executing in restricted execution mode; f_lasti gives the precise instruction (this is an index into the bytecode string of the code object).

Special writable attributes: f_trace, if not None, is a function called at the start of each source code line (this is used by the debugger); f_exc_type, f_exc_value, f_exc_traceback represent the last exception raised in the parent frame provided another exception was ever raised in the current
frame (in all other cases they are None); \texttt{f_lineno} is the current line number of the frame — writing to this from within a trace function jumps to the given line (only for the bottom-most frame). A debugger can implement a Jump command (aka Set Next Statement) by writing to \texttt{f_lineno}.

Traceback objects

Traceback objects represent a stack trace of an exception. A traceback object is created when an exception occurs. When the search for an exception handler unwinds the execution stack, at each unwound level a traceback object is inserted in front of the current traceback. When an exception handler is entered, the stack trace is made available to the program. (See section \texttt{The try statement}.) It is accessible as \texttt{sys.exc_traceback}, and also as the third item of the tuple returned by \texttt{sys.exc_info()}. The latter is the preferred interface, since it works correctly when the program is using multiple threads. When the program contains no suitable handler, the stack trace is written (nicely formatted) to the standard error stream; if the interpreter is interactive, it is also made available to the user as \texttt{sys.last_traceback}.

Special read-only attributes: \texttt{tb_next} is the next level in the stack trace (towards the frame where the exception occurred), or \texttt{None} if there is no next level; \texttt{tb_frame} points to the execution frame of the current level; \texttt{tb_lineno} gives the line number where the exception occurred; \texttt{tb_lasti} indicates the precise instruction. The line number and last instruction in the traceback may differ from the line number of its frame object if the exception occurred in a \texttt{try} statement with no matching except clause or with a finally clause.

Slice objects

Slice objects are used to represent slices when \textit{extended slice}
syntax is used. This is a slice using two colons, or multiple slices or ellipses separated by commas, e.g., \a[i:j:step], \a[i:j, k:l], or \a[... , i:j]. They are also created by the built-in slice() function.

Special read-only attributes: \texttt{start} is the lower bound; \texttt{stop} is the upper bound; \texttt{step} is the step value; each is \texttt{None} if omitted. These attributes can have any type.

Slice objects support one method:

\texttt{slice.indices(self, length)}

This method takes a single integer argument \texttt{length} and computes information about the extended slice that the slice object would describe if applied to a sequence of \texttt{length} items. It returns a tuple of three integers; respectively these are the \texttt{start} and \texttt{stop} indices and the \texttt{step} or stride length of the slice. Missing or out-of-bounds indices are handled in a manner consistent with regular slices.

\textit{New in version 2.3.}

Static method objects

Static method objects provide a way of defeating the transformation of function objects to method objects described above. A static method object is a wrapper around any other object, usually a user-defined method object. When a static method object is retrieved from a class or a class instance, the object actually returned is the wrapped object, which is not subject to any further transformation. Static method objects are not themselves callable, although the objects they wrap usually are. Static method objects are created by the built-in \texttt{staticmethod()} constructor.
Class method objects

A class method object, like a static method object, is a wrapper around another object that alters the way in which that object is retrieved from classes and class instances. The behaviour of class method objects upon such retrieval is described above, under “User-defined methods”. Class method objects are created by the built-in `classmethod()` constructor.
3.3. New-style and classic classes

Classes and instances come in two flavors: old-style (or classic) and new-style.

Up to Python 2.1 the concept of `class` was unrelated to the concept of `type`, and old-style classes were the only flavor available. For an old-style class, the statement `x.__class__` provides the class of `x`, but `type(x)` is always `<type 'instance'>`. This reflects the fact that all old-style instances, independent of their class, are implemented with a single built-in type, called `instance`.

New-style classes were introduced in Python 2.2 to unify the concepts of `class` and `type`. A new-style class is simply a user-defined type, no more, no less. If `x` is an instance of a new-style class, then `type(x)` is typically the same as `x.__class__` (although this is not guaranteed – a new-style class instance is permitted to override the value returned for `x.__class__`).

The major motivation for introducing new-style classes is to provide a unified object model with a full meta-model. It also has a number of practical benefits, like the ability to subclass most built-in types, or the introduction of “descriptors”, which enable computed properties.

For compatibility reasons, classes are still old-style by default. New-style classes are created by specifying another new-style class (i.e. a type) as a parent class, or the “top-level type” `object` if no other parent is needed. The behaviour of new-style classes differs from that of old-style classes in a number of important details in addition to what `type()` returns. Some of these changes are fundamental to the new object model, like the way special methods are invoked. Others are “fixes” that could not be implemented before for compatibility concerns, like the method resolution order in case of
multiple inheritance.

While this manual aims to provide comprehensive coverage of Python’s class mechanics, it may still be lacking in some areas when it comes to its coverage of new-style classes. Please see https://www.python.org/doc/newstyle/ for sources of additional information.

Old-style classes are removed in Python 3, leaving only new-style classes.
3.4. Special method names

A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. This is Python’s approach to operator overloading, allowing classes to define their own behavior with respect to language operators. For instance, if a class defines a method named `__getitem__()`, and `x` is an instance of this class, then `x[i]` is roughly equivalent to `x.__getitem__(i)` for old-style classes and `type(x).__getitem__(x, i)` for new-style classes. Except where mentioned, attempts to execute an operation raise an exception when no appropriate method is defined (typically `AttributeError` or `TypeError`).

When implementing a class that emulates any built-in type, it is important that the emulation only be implemented to the degree that it makes sense for the object being modelled. For example, some sequences may work well with retrieval of individual elements, but extracting a slice may not make sense. (One example of this is the `NodeList` interface in the W3C’s Document Object Model.)

3.4.1. Basic customization

```python
object.__new__(cls[, ...])
```

Called to create a new instance of class `cls`. `__new__()` is a static method (special-cased so you need not declare it as such) that takes the class of which an instance was requested as its first argument. The remaining arguments are those passed to the object constructor expression (the call to the class). The return value of `__new__()` should be the new object instance (usually an instance of `cls`).
Typical implementations create a new instance of the class by invoking the superclass's `__new__()` method using
```
super(currentclass, cls).__new__(cls[, ...])
```
with appropriate arguments and then modifying the newly-created instance as necessary before returning it.

If `__new__()` returns an instance of `cls`, then the new instance's `__init__()` method will be invoked like `__init__(self[, ...])`, where `self` is the new instance and the remaining arguments are the same as were passed to `__new__()`.

If `__new__()` does not return an instance of `cls`, then the new instance's `__init__()` method will not be invoked.

`__new__()` is intended mainly to allow subclasses of immutable types (like `int`, `str`, or `tuple`) to customize instance creation. It is also commonly overridden in custom metaclasses in order to customize class creation.

```
object.__init__(self[, ...])
```

Called after the instance has been created (by `__new__()`, but before it is returned to the caller. The arguments are those passed to the class constructor expression. If a base class has an `__init__()` method, the derived class's `__init__()` method, if any, must explicitly call it to ensure proper initialization of the base class part of the instance; for example:
```
BaseClass.__init__(self, [args...]).
```

Because `__new__()` and `__init__()` work together in constructing objects (`__new__()` to create it, and `__init__()` to customise it), no non-`None` value may be returned by `__init__()`; doing so will cause a `TypeError` to be raised at runtime.
object.__del__(self)

Called when the instance is about to be destroyed. This is also called a destructor. If a base class has a __del__() method, the derived class's __del__() method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance. Note that it is possible (though not recommended!) for the __del__() method to postpone destruction of the instance by creating a new reference to it. It may then be called at a later time when this new reference is deleted. It is not guaranteed that __del__() methods are called for objects that still exist when the interpreter exits.

**Note:** del x doesn't directly call x.__del__() — the former decrements the reference count for x by one, and the latter is only called when x's reference count reaches zero. Some common situations that may prevent the reference count of an object from going to zero include: circular references between objects (e.g., a doubly-linked list or a tree data structure with parent and child pointers); a reference to the object on the stack frame of a function that caught an exception (the traceback stored in sys.exc_traceback keeps the stack frame alive); or a reference to the object on the stack frame that raised an unhandled exception in interactive mode (the traceback stored in sys.last_traceback keeps the stack frame alive). The first situation can only be remedied by explicitly breaking the cycles; the latter two situations can be resolved by storing None in sys.exc_traceback or sys.last_traceback. Circular references which are garbage are detected when the option cycle detector is enabled (it's on by default), but can only be cleaned up if there are no Python-level __del__() methods involved. Refer to the documentation for the gc module for more information about how __del__() methods are handled by
the cycle detector, particularly the description of the garbage value.

**Warning:** Due to the precarious circumstances under which \_\_del\_\_() methods are invoked, exceptions that occur during their execution are ignored, and a warning is printed to `sys.stderr` instead. Also, when \_\_del\_\_() is invoked in response to a module being deleted (e.g., when execution of the program is done), other globals referenced by the \_\_del\_\_() method may already have been deleted or in the process of being torn down (e.g. the import machinery shutting down). For this reason, \_\_del\_\_() methods should do the absolute minimum needed to maintain external invariants. Starting with version 1.5, Python guarantees that globals whose name begins with a single underscore are deleted from their module before other globals are deleted; if no other references to such globals exist, this may help in assuring that imported modules are still available at the time when the \_\_del\_\_() method is called.

See also the `-R` command-line option.

**object.\_\_repr\_\_(self)**

Called by the `repr()` built-in function and by string conversions (reverse quotes) to compute the “official” string representation of an object. If at all possible, this should look like a valid Python expression that could be used to recreate an object with the same value (given an appropriate environment). If this is not possible, a string of the form `<...some useful description...>` should be returned. The return value must be a string object. If a class defines \_\_repr\_\_() but not \_\_str\_\_(), then \_\_repr\_\_() is also used when an “informal” string representation of instances of that class is required.
This is typically used for debugging, so it is important that the representation is information-rich and unambiguous.

```
object.__str__(self)
```

Called by the `str()` built-in function and by the `print` statement to compute the “informal” string representation of an object. This differs from `__repr__()` in that it does not have to be a valid Python expression: a more convenient or concise representation may be used instead. The return value must be a string object.

```
object.__lt__(self, other)
object.__le__(self, other)
object.__eq__(self, other)
object.__ne__(self, other)
object.__gt__(self, other)
object.__ge__(self, other)
```

*New in version 2.1.*

These are the so-called “rich comparison” methods, and are called for comparison operators in preference to `__cmp__()` below. The correspondence between operator symbols and method names is as follows: `x<y` calls `x.__lt__(y)`, `x<=y` calls `x.__le__(y)`, `x==y` calls `x.__eq__(y)`, `x!=y` and `x<>y` call `x.__ne__(y)`, `x>y` calls `x.__gt__(y)`, and `x>=y` calls `x.__ge__(y)`. A rich comparison method may return the singleton `NotImplemented` if it does not implement the operation for a given pair of arguments. By convention, `False` and `True` are returned for a successful comparison. However, these methods can return any value, so if the comparison operator is used in a Boolean context (e.g., in the condition of an `if` statement), Python will call `bool()` on the value to determine if the result is true or false.

There are no implied relationships among the comparison
operators. The truth of \( x==y \) does not imply that \( x!=y \) is false. Accordingly, when defining \( \text{__eq__}() \), one should also define \( \text{__ne__}() \) so that the operators will behave as expected. See the paragraph on \( \text{__hash__}() \) for some important notes on creating \textit{hashable} objects which support custom comparison operations and are usable as dictionary keys.

There are no swapped-argument versions of these methods (to be used when the left argument does not support the operation but the right argument does); rather, \( \text{__lt__}() \) and \( \text{__gt__}() \) are each other’s reflection, \( \text{__le__}() \) and \( \text{__ge__}() \) are each other’s reflection, and \( \text{__eq__}() \) and \( \text{__ne__}() \) are their own reflection.

Arguments to rich comparison methods are never coerced.

To automatically generate ordering operations from a single root operation, see \texttt{functools.total\_ordering()}.

\texttt{object.\_\_cmp\_\_(self, other)}

Called by comparison operations if rich comparison (see above) is not defined. Should return a negative integer if \texttt{self < other}, zero if \texttt{self == other}, a positive integer if \texttt{self > other}. If no \( \text{__cmp__}() \), \( \text{__eq__}() \) or \( \text{__ne__}() \) operation is defined, class instances are compared by object identity ("address"). See also the description of \( \text{__hash__}() \) for some important notes on creating \textit{hashable} objects which support custom comparison operations and are usable as dictionary keys. (Note: the restriction that exceptions are not propagated by \( \text{__cmp__}() \) has been removed since Python 1.5.)

\texttt{object.\_\_rcmp\_\_(self, other)}

\textit{Changed in version 2.1}: No longer supported.

\texttt{object.\_\_hash\_\_(self)}
Called by built-in function `hash()` and for operations on members of hashed collections including `set`, `frozenset`, and `dict`. `__hash__()` should return an integer. The only required property is that objects which compare equal have the same hash value; it is advised to somehow mix together (e.g. using exclusive or) the hash values for the components of the object that also play a part in comparison of objects.

If a class does not define a `__cmp__()` or `__eq__()` method it should not define a `__hash__()` operation either; if it defines `__cmp__()` or `__eq__()` but not `__hash__()`, its instances will not be usable in hashed collections. If a class defines mutable objects and implements a `__cmp__()` or `__eq__()` method, it should not implement `__hash__()`, since hashable collection implementations require that an object's hash value is immutable (if the object's hash value changes, it will be in the wrong hash bucket).

User-defined classes have `__cmp__()` and `__hash__()` methods by default; with them, all objects compare unequal (except with themselves) and `x.__hash__()` returns a result derived from `id(x)`.

Classes which inherit a `__hash__()` method from a parent class but change the meaning of `__cmp__()` or `__eq__()` such that the hash value returned is no longer appropriate (e.g. by switching to a value-based concept of equality instead of the default identity based equality) can explicitly flag themselves as being unhashable by setting `__hash__ = None` in the class definition. Doing so means that not only will instances of the class raise an appropriate `TypeError` when a program attempts to retrieve their hash value, but they will also be correctly identified as unhashable when checking `isinstance(obj,
collections.Hashable) (unlike classes which define their own __hash__() to explicitly raise TypeError).

*Changed in version 2.5:* __hash__() may now also return a long integer object; the 32-bit integer is then derived from the hash of that object.

*Changed in version 2.6:* __hash__ may now be set to None to explicitly flag instances of a class as unhashable.

**object.__nonzero__(self)**
Called to implement truth value testing and the built-in operation bool(); should return False or True, or their integer equivalents 0 or 1. When this method is not defined, __len__() is called, if it is defined, and the object is considered true if its result is nonzero. If a class defines neither __len__() nor __nonzero__(), all its instances are considered true.

**object.__unicode__(self)**
Called to implement unicode() built-in; should return a Unicode object. When this method is not defined, string conversion is attempted, and the result of string conversion is converted to Unicode using the system default encoding.

### 3.4.2. Customizing attribute access

The following methods can be defined to customize the meaning of attribute access (use of, assignment to, or deletion of x.name) for class instances.

**object.__getattr__(self, name)**
Called when an attribute lookup has not found the attribute in the usual places (i.e. it is not an instance attribute nor is it found in the class tree for self). name is the attribute name. This method
should return the (computed) attribute value or raise an AttributeError exception.

Note that if the attribute is found through the normal mechanism, \_\_getattr\_() is not called. (This is an intentional asymmetry between \_\_getattr\_() and \_\_setattr\_().) This is done both for efficiency reasons and because otherwise \_\_getattr\_() would have no way to access other attributes of the instance. Note that at least for instance variables, you can fake total control by not inserting any values in the instance attribute dictionary (but instead inserting them in another object). See the \_\_getattr\_() method below for a way to actually get total control in new-style classes.

object.\_\_setattr\_(self, name, value)

Called when an attribute assignment is attempted. This is called instead of the normal mechanism (i.e. store the value in the instance dictionary). name is the attribute name, value is the value to be assigned to it.

If \_\_setattr\_() wants to assign to an instance attribute, it should not simply execute self.name = value — this would cause a recursive call to itself. Instead, it should insert the value in the dictionary of instance attributes, e.g., self.\_\_dict\_[name] = value. For new-style classes, rather than accessing the instance dictionary, it should call the base class method with the same name, for example, object.\_\_setattr\_(self, name, value).

object.\_\_delattr\_(self, name)

Like \_\_setattr\_() but for attribute deletion instead of assignment. This should only be implemented if del obj.name is meaningful for the object.

3.4.2.1. More attribute access for new-style classes
The following methods only apply to new-style classes.

```python
object.__getattribute__(self, name)
```

Called unconditionally to implement attribute accesses for instances of the class. If the class also defines `__getattr__`, the latter will not be called unless `__getattribute__` either calls it explicitly or raises an `AttributeError`. This method should return the (computed) attribute value or raise an `AttributeError` exception. In order to avoid infinite recursion in this method, its implementation should always call the base class method with the same name to access any attributes it needs, for example, `object.__getattribute__(self, name)`.

**Note:** This method may still be bypassed when looking up special methods as the result of implicit invocation via language syntax or built-in functions. See [Special method lookup for new-style classes](#).

### 3.4.2.2. Implementing Descriptors

The following methods only apply when an instance of the class containing the method (a so-called descriptor class) appears in an owner class (the descriptor must be in either the owner’s class dictionary or in the class dictionary for one of its parents). In the examples below, “the attribute” refers to the attribute whose name is the key of the property in the owner class’ `_dict_`.

```python
object.__get__(self, instance, owner)
```

Called to get the attribute of the owner class (class attribute access) or of an instance of that class (instance attribute access). `owner` is always the owner class, while `instance` is the instance that the attribute was accessed through, or `None` when the attribute is accessed through the `owner`. This method should
return the (computed) attribute value or raise an `AttributeError` exception.

```python
object.__set__(self, instance, value)
```

Called to set the attribute on an instance `instance` of the owner class to a new value, `value`.

```python
object.__delete__(self, instance)
```

Called to delete the attribute on an instance `instance` of the owner class.

### 3.4.2.3. Invoking Descriptors

In general, a descriptor is an object attribute with “binding behavior”, one whose attribute access has been overridden by methods in the descriptor protocol: `__get__`, `__set__`, and `__delete__`. If any of those methods are defined for an object, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object’s dictionary. For instance, `a.x` has a lookup chain starting with `a.__dict__['x']`, then `type(a).__dict__['x']`, and continuing through the base classes of `type(a)` excluding metaclasses.

However, if the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined and how they were called. Note that descriptors are only invoked for new style objects or classes (ones that subclass `object()` or `type()`).

The starting point for descriptor invocation is a binding, `a.x`. How the
arguments are assembled depends on a:

Direct Call
The simplest and least common call is when user code directly invokes a descriptor method: \texttt{x.__get__(a)}.

Instance Binding
If binding to a new-style object instance, \texttt{a.x} is transformed into the call: \texttt{type(a).__dict__['x'].\_get(a, type(a))}.

Class Binding
If binding to a new-style class, \texttt{A.x} is transformed into the call: \texttt{A.__dict__['x'].\_get(None, A)}.

Super Binding
If \texttt{a} is an instance of \texttt{super}, then the binding \texttt{super(B, obj).m()} searches \texttt{obj.__class__.__mro__} for the base class \texttt{A} immediately preceding \texttt{B} and then invokes the descriptor with the call: \texttt{A.__dict__['m'].\_get(obj, obj.__class__)}.  

For instance bindings, the precedence of descriptor invocation depends on the which descriptor methods are defined. A descriptor can define any combination of \texttt{\_get()}, \texttt{\_set()} and \texttt{\_delete()}. If it does not define \texttt{\_get()}, then accessing the attribute will return the descriptor object itself unless there is a value in the object’s instance dictionary. If the descriptor defines \texttt{\_set()} and/or \texttt{\_delete()}, it is a data descriptor; if it defines neither, it is a non-data descriptor. Normally, data descriptors define both \texttt{\_get()} and \texttt{\_set()}, while non-data descriptors have just the \texttt{\_get()} method. Data descriptors with \texttt{\_set()} and \texttt{\_get()} defined always override a redefinition in an instance dictionary. In contrast, non-data descriptors can be overridden by instances.

Python methods (including \texttt{staticmethod()} and \texttt{classmethod()}) are implemented as non-data descriptors. Accordingly, instances can
redefine and override methods. This allows individual instances to acquire behaviors that differ from other instances of the same class.

The `property()` function is implemented as a data descriptor. Accordingly, instances cannot override the behavior of a property.

3.4.2.4. __slots__

By default, instances of both old and new-style classes have a dictionary for attribute storage. This wastes space for objects having very few instance variables. The space consumption can become acute when creating large numbers of instances.

The default can be overridden by defining `__slots__` in a new-style class definition. The `__slots__` declaration takes a sequence of instance variables and reserves just enough space in each instance to hold a value for each variable. Space is saved because `__dict__` is not created for each instance.

`__slots__`

This class variable can be assigned a string, iterable, or sequence of strings with variable names used by instances. If defined in a new-style class, `__slots__` reserves space for the declared variables and prevents the automatic creation of `__dict__` and `__weakref__` for each instance.

*New in version 2.2.*

Notes on using `__slots__`

- When inheriting from a class without `__slots__`, the `__dict__` attribute of that class will always be accessible, so a `__slots__` definition in the subclass is meaningless.

- Without a `__dict__` variable, instances cannot be assigned new variables not listed in the `__slots__` definition. Attempts to
assign to an unlisted variable name raises `AttributeError`. If dynamic assignment of new variables is desired, then add '__dict__' to the sequence of strings in the '__slots__' declaration.

*Changed in version 2.3:* Previously, adding '__dict__' to the '__slots__' declaration would not enable the assignment of new attributes not specifically listed in the sequence of instance variable names.

- Without a '__weakref__' variable for each instance, classes defining '__slots__' do not support weak references to its instances. If weak reference support is needed, then add '__weakref__' to the sequence of strings in the '__slots__' declaration.

*Changed in version 2.3:* Previously, adding '__weakref__' to the '__slots__' declaration would not enable support for weak references.

- '__slots__' are implemented at the class level by creating descriptors ([Implementing Descriptors](Implementing Descriptors)) for each variable name. As a result, class attributes cannot be used to set default values for instance variables defined by '__slots__; otherwise, the class attribute would overwrite the descriptor assignment.

- The action of a '__slots__' declaration is limited to the class where it is defined. As a result, subclasses will have a '__dict__' unless they also define '__slots__' (which must only contain names of any *additional* slots).

- If a class defines a slot also defined in a base class, the instance variable defined by the base class slot is inaccessible (except by retrieving its descriptor directly from the base class). This renders the meaning of the program undefined. In the
future, a check may be added to prevent this.

- Nonempty __slots__ does not work for classes derived from “variable-length” built-in types such as long, str and tuple.

- Any non-string iterable may be assigned to __slots__. Mappings may also be used; however, in the future, special meaning may be assigned to the values corresponding to each key.

- __class__ assignment works only if both classes have the same __slots__.

*Changed in version 2.6:* Previously, __class__ assignment raised an error if either new or old class had __slots__.

### 3.4.3. Customizing class creation

By default, new-style classes are constructed using type(). A class definition is read into a separate namespace and the value of class name is bound to the result of type(name, bases, dict).

When the class definition is read, if __metaclass__ is defined then the callable assigned to it will be called instead of type(). This allows classes or functions to be written which monitor or alter the class creation process:

- Modifying the class dictionary prior to the class being created.
- Returning an instance of another class – essentially performing the role of a factory function.

These steps will have to be performed in the metaclass's __new__() method – type.__new__() can then be called from this method to create a class with different properties. This example adds a new element to the class dictionary before creating the class:
class metacls(type):
    def __new__(mcs, name, bases, dict):
        dict['foo'] = 'metacls was here'
        return type.__new__(mcs, name, bases, dict)

You can of course also override other class methods (or add new methods); for example defining a custom __call__() method in the metaclass allows custom behavior when the class is called, e.g. not always creating a new instance.

__metaclass__

This variable can be any callable accepting arguments for name, bases, and dict. Upon class creation, the callable is used instead of the built-in type().

New in version 2.2.

The appropriate metaclass is determined by the following precedence rules:

- If dict['__metaclass__'] exists, it is used.
- Otherwise, if there is at least one base class, its metaclass is used (this looks for a __class__ attribute first and if not found, uses its type).
- Otherwise, if a global variable named __metaclass__ exists, it is used.
- Otherwise, the old-style, classic metaclass (types.ClassType) is used.

The potential uses for metaclasses are boundless. Some ideas that have been explored including logging, interface checking, automatic delegation, automatic property creation, proxies, frameworks, and automatic resource locking/synchronization.

3.4.4. Customizing instance and subclass checks
New in version 2.6.

The following methods are used to override the default behavior of the `isinstance()` and `issubclass()` built-in functions.

In particular, the metaclass `abc.ABCMeta` implements these methods in order to allow the addition of Abstract Base Classes (ABCs) as “virtual base classes” to any class or type (including built-in types), including other ABCs.

```python
class.__instancecheck__(self, instance)
    Return true if instance should be considered a (direct or indirect) instance of class. If defined, called to implement `isinstance(instance, class)`.

class.__subclasscheck__(self, subclass)
    Return true if subclass should be considered a (direct or indirect) subclass of class. If defined, called to implement `issubclass(subclass, class)`.
```

Note that these methods are looked up on the type (metaclass) of a class. They cannot be defined as class methods in the actual class. This is consistent with the lookup of special methods that are called on instances, only in this case the instance is itself a class.

See also:

**PEP 3119 - Introducing Abstract Base Classes**
Includes the specification for customizing `isinstance()` and `issubclass()` behavior through `__instancecheck__()` and `__subclasscheck__()`, with motivation for this functionality in the context of adding Abstract Base Classes (see the `abc` module) to the language.
3.4.5. Emulating callable objects

object.__call__(self, args...)

Called when the instance is "called" as a function; if this method is defined, x(arg1, arg2, ...) is a shorthand for x.__call__(arg1, arg2, ...).

3.4.6. Emulating container types

The following methods can be defined to implement container objects. Containers usually are sequences (such as lists or tuples) or mappings (like dictionaries), but can represent other containers as well. The first set of methods is used either to emulate a sequence or to emulate a mapping; the difference is that for a sequence, the allowable keys should be the integers \( k \) for which \( 0 \leq k < N \) where \( N \) is the length of the sequence, or slice objects, which define a range of items. (For backwards compatibility, the method __getslice__() (see below) can also be defined to handle simple, but not extended slices.) It is also recommended that mappings provide the methods keys(), values(), items(), has_key(), get(), clear(), setdefault(), iterkeys(), itervalues(), iteritems(), pop(), popitem(), copy(), and update() behaving similar to those for Python's standard dictionary objects. The UserDict module provides a DictMixin class to help create those methods from a base set of __getitem__(), __setitem__(), __delitem__(), and keys(). Mutable sequences should provide methods append(), count(), index(), extend(), insert(), pop(), remove(), reverse() and sort(), like Python standard list objects. Finally, sequence types should implement addition (meaning concatenation) and multiplication (meaning repetition) by defining the methods __add__(), __radd__(), __iadd__(), __mul__(), __rmul__() and __imul__() described below; they should not define __coerce__() or other numerical operators. It
is recommended that both mappings and sequences implement the __contains__() method to allow efficient use of the in operator; for mappings, in should be equivalent of has_key(); for sequences, it should search through the values. It is further recommended that both mappings and sequences implement the __iter__() method to allow efficient iteration through the container; for mappings, __iter__() should be the same as iterkeys(); for sequences, it should iterate through the values.

object.__len__(self)
Called to implement the built-in function len(). Should return the length of the object, an integer >= 0. Also, an object that doesn’t define a __nonzero__() method and whose __len__() method returns zero is considered to be false in a Boolean context.

object.__getitem__(self, key)
Called to implement evaluation of self[key]. For sequence types, the accepted keys should be integers and slice objects. Note that the special interpretation of negative indexes (if the class wishes to emulate a sequence type) is up to the __getitem__() method. If key is of an inappropriate type, TypeError may be raised; if of a value outside the set of indexes for the sequence (after any special interpretation of negative values), IndexError should be raised. For mapping types, if key is missing (not in the container), KeyError should be raised.

Note: for loops expect that an IndexError will be raised for illegal indexes to allow proper detection of the end of the sequence.

object.__missing__(self, key)
Called by dict.__getitem__() to implement self[key] for dict subclasses when key is not in the dictionary.
object.__setitem__(self, key, value)

Called to implement assignment to self[key]. Same note as for __getitem__(). This should only be implemented for mappings if the objects support changes to the values for keys, or if new keys can be added, or for sequences if elements can be replaced. The same exceptions should be raised for improper key values as for the __getitem__() method.

object.__delitem__(self, key)

Called to implement deletion of self[key]. Same note as for __getitem__(). This should only be implemented for mappings if the objects support removal of keys, or for sequences if elements can be removed from the sequence. The same exceptions should be raised for improper key values as for the __getitem__() method.

object.__iter__(self)

This method is called when an iterator is required for a container. This method should return a new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container, and should also be made available as the method iterkeys().

Iterator objects also need to implement this method; they are required to return themselves. For more information on iterator objects, see Iterator Types.

object.__reversed__(self)

Called (if present) by the reversed() built-in to implement reverse iteration. It should return a new iterator object that iterates over all the objects in the container in reverse order.

If the __reversed__() method is not provided, the reversed() built-in will fall back to using the sequence protocol (__len__()...
and \texttt{\_\_getitem\_()}). Objects that support the sequence protocol should only provide \texttt{\_\_reversed\_()} if they can provide an implementation that is more efficient than the one provided by \texttt{reversed()}. 

\textit{New in version 2.6.}

The membership test operators (\texttt{in} and \texttt{not in}) are normally implemented as an iteration through a sequence. However, container objects can supply the following special method with a more efficient implementation, which also does not require the object be a sequence.

\texttt{object.\_\_contains\_\_}(\texttt{self, item})

Called to implement membership test operators. Should return true if \texttt{item} is in \texttt{self}, false otherwise. For mapping objects, this should consider the keys of the mapping rather than the values or the key-item pairs.

For objects that don’t define \texttt{\_\_contains\_()}, the membership test first tries iteration via \texttt{\_\_iter\_()}, then the old sequence iteration protocol via \texttt{\_\_getitem\_()}, see this section in the language reference.

\subsection*{3.4.7. Additional methods for emulation of sequence types}

The following optional methods can be defined to further emulate sequence objects. Immutable sequences methods should at most only define \texttt{\_\_getslice\_()}; mutable sequences might define all three methods.

\texttt{object.\_\_getslice\_\_}(\texttt{self, i, j})

\textit{Deprecated since version 2.0: Support slice objects as}
parameters to the \_\_getitem\_\_ method. (However, built-in
types in CPython currently still implement \_\_getslice\_\_. Therefore, you have to override it in derived classes when
implementing slicing.)

Called to implement evaluation of \texttt{self[i:j]}. The returned object
should be of the same type as \texttt{self}. Note that missing \texttt{i} or \texttt{j} in the
slice expression are replaced by zero or \texttt{sys.maxsize},
respectively. If negative indexes are used in the slice, the length
of the sequence is added to that index. If the instance does not
implement the \_\_len\_\_ method, an \texttt{AttributeError} is raised.
No guarantee is made that indexes adjusted this way are not still
negative. Indexes which are greater than the length of the
sequence are not modified. If no \_\_getslice\_\_ is found, a slice
object is created instead, and passed to \_\_getitem\_\_ instead.

\texttt{object.\_\_setslice\_\_(self, i, j, sequence)}

Called to implement assignment to \texttt{self[i:j]}. Same notes for \texttt{i}
and \texttt{j} as for \_\_getslice\_\_.

This method is deprecated. If no \_\_setslice\_\_ is found, or for
extended slicing of the form \texttt{self[i:j:k]}, a slice object is
created, and passed to \_\_setitem\_\_, instead of \_\_setslice\_\_
being called.

\texttt{object.\_\_delslice\_\_(self, i, j)}

Called to implement deletion of \texttt{self[i:j]}. Same notes for \texttt{i}
and \texttt{j} as for \_\_getslice\_\_. This method is deprecated. If no
\_\_delslice\_\_ is found, or for extended slicing of the form
\texttt{self[i:j:k]}, a slice object is created, and passed to
\_\_delitem\_\_, instead of \_\_delslice\_\_ being called.

Notice that these methods are only invoked when a single slice with
a single colon is used, and the slice method is available. For slice
operations involving extended slice notation, or in absence of the slice methods, \_\_getitem\_\_, \_\_setitem\_\_ or \_\_delitem\_\_ is called with a slice object as argument.

The following example demonstrate how to make your program or module compatible with earlier versions of Python (assuming that methods \_\_getitem\_\_, \_\_setitem\_\_ and \_\_delitem\_\_ support slice objects as arguments):

```python
class MyClass:
    ...
    def \_\_getitem\_\_(self, index):
        ...
    def \_\_setitem\_\_(self, index, value):
        ...
    def \_\_delitem\_\_(self, index):
        ...

    if sys.version_info < (2, 0):
        # They won't be defined if version is at least 2.0 final
        def \_\_getslice\_\_(self, i, j):
            return self[max(0, i):max(0, j):]
        def \_\_setslice\_\_(self, i, j, seq):
            self[max(0, i):max(0, j):] = seq
        def \_\_delslice\_\_(self, i, j):
            del self[max(0, i):max(0, j):]
    ...
```

Note the calls to \texttt{max}(); these are necessary because of the handling of negative indices before the \_\_*\_\_slice\_\_() methods are called. When negative indexes are used, the \_\_*\_\_item\_\_() methods receive them as provided, but the \_\_*\_\_slice\_\_() methods get a “cooked” form of the index values. For each negative index value, the length of the sequence is added to the index before calling the method (which may still result in a negative index); this is the customary handling of negative indexes by the built-in sequence types, and the \_\_*\_\_item\_\_() methods are expected to do this as well. However,
since they should already be doing that, negative indexes cannot be passed in; they must be constrained to the bounds of the sequence before being passed to the `__getitem__()` methods. Calling `max(0, i)` conveniently returns the proper proper value.

### 3.4.8. Emulating numeric types

The following methods can be defined to emulate numeric objects. Methods corresponding to operations that are not supported by the particular kind of number implemented (e.g., bitwise operations for non-integral numbers) should be left undefined.

```python
object.__add__(self, other)
object.__sub__(self, other)
object.__mul__(self, other)
object.__floordiv__(self, other)
object.__mod__(self, other)
object.__divmod__(self, other)
object.__pow__(self, other[, modulo])
object.__lshift__(self, other)
object.__rshift__(self, other)
object.__and__(self, other)
object.__xor__(self, other)
object.__or__(self, other)
```

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `/`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`). For instance, to evaluate the expression `x + y`, where `x` is an instance of a class that has an `__add__()` method, `x.__add__(y)` is called. The `__divmod__()` method should be the equivalent to using `__floordiv__()` and `__mod__(); it should not be related to `__truediv__()` (described below). Note that `__pow__()` should be defined to accept an optional third argument if the ternary version of the built-in `pow()` function is to be supported.
If one of those methods does not support the operation with the supplied arguments, it should return `NotImplemented`.

```python
object.__div__(self, other)
object.__truediv__(self, other)
```

The division operator (`/`) is implemented by these methods. The `__truediv__()` method is used when `__future__.division` is in effect, otherwise `__div__()` is used. If only one of these two methods is defined, the object will not support division in the alternate context; `TypeError` will be raised instead.

```python
object.__radd__(self, other)
object.__rsub__(self, other)
object.__rmul__(self, other)
object.__rdiv__(self, other)
object.__rtruediv__(self, other)
object.__rfloordiv__(self, other)
object.__rmod__(self, other)
object.__rdivmod__(self, other)
object.__rpow__(self, other)
object.__rlshift__(self, other)
object.__rrshift__(self, other)
object.__rand__(self, other)
object.__rxor__(self, other)
object.__ror__(self, other)
```

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `/`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation and the operands are of different types. [2] For instance, to evaluate the expression `x - y`, where `y` is an instance of a class that has an `__rsub__()` method, `y.__rsub__(x)` is called if `x.__sub__(y)` returns `NotImplemented`.

Note that ternary `pow()` will not try calling `__rpow__()`) (the
coercion rules would become too complicated).

**Note:** If the right operand's type is a subclass of the left operand's type and that subclass provides the reflected method for the operation, this method will be called before the left operand's non-reflected method. This behavior allows subclasses to override their ancestors' operations.

```python
object.__iadd__(self, other)
object.__isub__(self, other)
object.__imul__(self, other)
object.__idiv__(self, other)
object.__itruediv__(self, other)
object.__ifloordiv__(self, other)
object.__imod__(self, other)
object.__ipow__(self, other[
    , modulo]
)
object.__ilshift__(self, other)
object.__irshift__(self, other)
object.__iand__(self, other)
object.__ixor__(self, other)
object.__ior__(self, other)
```

These methods are called to implement the augmented arithmetic assignments (+=, -=, *=, /=, //=, %=, **=, <<=, >>=, &=, ^=, |=). These methods should attempt to do the operation in-place (modifying `self`) and return the result (which could be, but does not have to be, `self`). If a specific method is not defined, the augmented assignment falls back to the normal methods. For instance, to execute the statement `x += y`, where `x` is an instance of a class that has an `__iadd__()` method, `x.__iadd__(y)` is called. If `x` is an instance of a class that does not define a `__iadd__()` method, `x.__add__(y)` and `y.__radd__(x)` are considered, as with the evaluation of `x + y`.

```python
object.__neg__(self)
```
object.__pos__(self)
object.__abs__(self)
object.__invert__(self)
    Called to implement the unary arithmetic operations (+, - and ~).

object.__complex__(self)
object.__int__(self)
object.__long__(self)
object.__float__(self)
    Called to implement the built-in functions complex(), int(), long(), and float(). Should return a value of the appropriate type.

object.__oct__(self)
object.__hex__(self)
    Called to implement the built-in functions oct() and hex(). Should return a string value.

object.__index__(self)
    Called to implement operator.index(). Also called whenever Python needs an integer object (such as in slicing). Must return an integer (int or long).

    New in version 2.5.

object.__coerce__(self, other)
    Called to implement “mixed-mode” numeric arithmetic. Should either return a 2-tuple containing self and other converted to a common numeric type, or None if conversion is impossible. When the common type would be the type of other, it is sufficient to return None, since the interpreter will also ask the other object to attempt a coercion (but sometimes, if the implementation of the other type cannot be changed, it is useful to do the conversion to
the other type here). A return value of `NotImplemented` is equivalent to returning `None`.

3.4.9. Coercion rules

This section used to document the rules for coercion. As the language has evolved, the coercion rules have become hard to document precisely; documenting what one version of one particular implementation does is undesirable. Instead, here are some informal guidelines regarding coercion. In Python 3, coercion will not be supported.

- If the left operand of a `%` operator is a string or Unicode object, no coercion takes place and the string formatting operation is invoked instead.

- It is no longer recommended to define a coercion operation. Mixed-mode operations on types that don’t define coercion pass the original arguments to the operation.

- New-style classes (those derived from `object`) never invoke the `__coerce__()` method in response to a binary operator; the only time `__coerce__()` is invoked is when the built-in function `coerce()` is called.

- For most intents and purposes, an operator that returns `NotImplemented` is treated the same as one that is not implemented at all.

- Below, `__op__()` and `__rop__()` are used to signify the generic method names corresponding to an operator; `__iop__()` is used for the corresponding in-place operator. For example, for the operator `'+'`, `__add__()` and `__radd__()` are used for the left and right variant of the binary operator, and `__iadd__()` for the in-
place variant.

- For objects $x$ and $y$, first `x.__op__(y)` is tried. If this is not implemented or returns `NotImplemented`, `y.__rop__(x)` is tried. If this is also not implemented or returns `NotImplemented`, a `TypeError` exception is raised. But see the following exception:

- Exception to the previous item: if the left operand is an instance of a built-in type or a new-style class, and the right operand is an instance of a proper subclass of that type or class and overrides the base’s `__rop__()` method, the right operand’s `__rop__()` method is tried before the left operand’s `__op__()` method.

This is done so that a subclass can completely override binary operators. Otherwise, the left operand’s `__op__()` method would always accept the right operand: when an instance of a given class is expected, an instance of a subclass of that class is always acceptable.

- When either operand type defines a coercion, this coercion is called before that type’s `__op__()` or `__rop__()` method is called, but no sooner. If the coercion returns an object of a different type for the operand whose coercion is invoked, part of the process is redone using the new object.

- When an in-place operator (like `+=`) is used, if the left operand implements `__iop__()`, it is invoked without any coercion. When the operation falls back to `__op__()` and/or `__rop__()`, the normal coercion rules apply.

- In `$x + y$`, if $x$ is a sequence that implements sequence concatenation, sequence concatenation is invoked.
• In \( x^*y \), if one operand is a sequence that implements sequence repetition, and the other is an integer (\texttt{int} or \texttt{long}), sequence repetition is invoked.

• Rich comparisons (implemented by methods \texttt{__eq__()} and so on) never use coercion. Three-way comparison (implemented by \texttt{__cmp__()}) does use coercion under the same conditions as other binary operations use it.

• In the current implementation, the built-in numeric types \texttt{int}, \texttt{long}, \texttt{float}, and \texttt{complex} do not use coercion. All these types implement a \texttt{__coerce__()} method, for use by the built-in \texttt{coerce()} function.

\textit{Changed in version 2.7:} The complex type no longer makes implicit calls to the \texttt{__coerce__()} method for mixed-type binary arithmetic operations.

### 3.4.10. With Statement Context Managers

\textit{New in version 2.5.}

A \textit{context manager} is an object that defines the runtime context to be established when executing a \texttt{with} statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. Context managers are normally invoked using the \texttt{with} statement (described in section The \texttt{with} statement), but can also be used by directly invoking their methods.

Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc.
For more information on context managers, see Context Manager Types.

object.__enter__(self)

Enter the runtime context related to this object. The with statement will bind this method’s return value to the target(s) specified in the as clause of the statement, if any.

object.__exit__(self, exc_type, exc_value, traceback)

Exit the runtime context related to this object. The parameters describe the exception that caused the context to be exited. If the context was exited without an exception, all three arguments will be None.

If an exception is supplied, and the method wishes to suppress the exception (i.e., prevent it from being propagated), it should return a true value. Otherwise, the exception will be processed normally upon exit from this method.

Note that __exit__() methods should not reraise the passed-in exception; this is the caller’s responsibility.

See also:

PEP 343 - The “with” statement

The specification, background, and examples for the Python with statement.

3.4.11. Special method lookup for old-style classes

For old-style classes, special methods are always looked up in exactly the same way as any other method or attribute. This is the case regardless of whether the method is being looked up explicitly as in x.__getitem__(i) or implicitly as in x[i].
This behaviour means that special methods may exhibit different behaviour for different instances of a single old-style class if the appropriate special attributes are set differently:

```python
>>> class C:
    ...
    pass
    ...
>>> c1 = C()
>>> c2 = C()
>>> c1.__len__ = lambda: 5
>>> c2.__len__ = lambda: 9
>>> len(c1)
5
>>> len(c2)
9
```

### 3.4.12. Special method lookup for new-style classes

For new-style classes, implicit invocations of special methods are only guaranteed to work correctly if defined on an object's type, not in the object's instance dictionary. That behaviour is the reason why the following code raises an exception (unlike the equivalent example with old-style classes):

```python
>>> class C(object):
    ...
    pass
    ...
>>> c = C()
>>> c.__len__ = lambda: 5
>>> len(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'C' has no len()
```

The rationale behind this behaviour lies with a number of special methods such as `__hash__()` and `__repr__()` that are implemented by all objects, including type objects. If the implicit lookup of these methods used the conventional lookup process, they would fail when
invoked on the type object itself:

```python
>>> 1.__hash__() == hash(1)
True
>>> int.__hash__() == hash(int)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor '__hash__' of 'int' object needs an argument
```

Incorrectly attempting to invoke an unbound method of a class in this way is sometimes referred to as ‘metaclass confusion’, and is avoided by bypassing the instance when looking up special methods:

```python
>>> type(1).__hash__(1) == hash(1)
True
>>> type(int).__hash__(int) == hash(int)
True
```

In addition to bypassing any instance attributes in the interest of correctness, implicit special method lookup generally also bypasses the `__getattr__()` method even of the object’s metaclass:

```python
>>> class Meta(type):
...    def __getattribute__(*args):
...        print "Metaclass getattribute invoked"
...        return type.__getattribute__(*args)
...    
>>> class C(object):
...        __metaclass__ = Meta
...        def __len__(self):
...            return 10
...        def __getattribute__(*args):
...            print "Class getattribute invoked"
...            return object.__getattribute__(*args)
...
>>> c = C()
>>> c.__len__()  # Explicit lookup via instance
Class getattribute invoked
10
>>> type(c).__len__(c)  # Explicit lookup via type
```
Bypassing the `__getattribute__()` machinery in this fashion provides significant scope for speed optimisations within the interpreter, at the cost of some flexibility in the handling of special methods (the special method `must` be set on the class object itself in order to be consistently invoked by the interpreter).

Footnotes

[1] It is possible in some cases to change an object’s type, under certain controlled conditions. It generally isn’t a good idea though, since it can lead to some very strange behaviour if it is handled incorrectly.

[2] For operands of the same type, it is assumed that if the non-reflected method (such as `__add__()`) fails the operation is not supported, which is why the reflected method is not called.
4. Execution model
4.1. Naming and binding

*Names* refer to objects. Names are introduced by name binding operations. Each occurrence of a name in the program text refers to the *binding* of that name established in the innermost function block containing the use.

A *block* is a piece of Python program text that is executed as a unit. The following are blocks: a module, a function body, and a class definition. Each command typed interactively is a block. A script file (a file given as standard input to the interpreter or specified on the interpreter command line the first argument) is a code block. A script command (a command specified on the interpreter command line with the ‘*-c*’ option) is a code block. The file read by the built-in function `execfile()` is a code block. The string argument passed to the built-in function `eval()` and to the `exec` statement is a code block. The expression read and evaluated by the built-in function `input()` is a code block.

A code block is executed in an *execution frame*. A frame contains some administrative information (used for debugging) and determines where and how execution continues after the code block’s execution has completed.

A *scope* defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. If the definition occurs in a function block, the scope extends to any blocks contained within the defining one, unless a contained block introduces a different binding for the name. The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods – this includes generator expressions since they are implemented using a function scope. This means that the following will fail:
class A:
    a = 42
    b = list(a + i for i in range(10))

When a name is used in a code block, it is resolved using the nearest enclosing scope. The set of all such scopes visible to a code block is called the block's environment.

If a name is bound in a block, it is a local variable of that block. If a name is bound at the module level, it is a global variable. (The variables of the module code block are local and global.) If a variable is used in a code block but not defined there, it is a free variable.

When a name is not found at all, a NameError exception is raised. If the name refers to a local variable that has not been bound, a UnboundLocalError exception is raised. UnboundLocalError is a subclass of NameError.

The following constructs bind names: formal parameters to functions, import statements, class and function definitions (these bind the class or function name in the defining block), and targets that are identifiers if occurring in an assignment, for loop header, in the second position of an except clause header or after as in a with statement. The import statement of the form from ... import * binds all names defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.

A target occurring in a del statement is also considered bound for this purpose (though the actual semantics are to unbind the name). It is illegal to unbind a name that is referenced by an enclosing scope; the compiler will report a SyntaxError.

Each assignment or import statement occurs within a block defined by a class or function definition or at the module level (the top-level
If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block. This can lead to errors when a name is used within a block before it is bound. This rule is subtle. Python lacks declarations and allows name binding operations to occur anywhere within a code block. The local variables of a code block can be determined by scanning the entire text of the block for name binding operations.

If the global statement occurs within a block, all uses of the name specified in the statement refer to the binding of that name in the top-level namespace. Names are resolved in the top-level namespace by searching the global namespace, i.e. the namespace of the module containing the code block, and the builtins namespace, the namespace of the module __builtins__. The global namespace is searched first. If the name is not found there, the builtins namespace is searched. The global statement must precede all uses of the name.

The builtins namespace associated with the execution of a code block is actually found by looking up the name __builtins__ in its global namespace; this should be a dictionary or a module (in the latter case the module’s dictionary is used). By default, when in the __main__ module, __builtins__ is the built-in module __builtin__ (note: no ‘s’); when in any other module, __builtins__ is an alias for the dictionary of the __builtin__ module itself. __builtins__ can be set to a user-created dictionary to create a weak form of restricted execution.

**CPython implementation detail:** Users should not touch __builtins__; it is strictly an implementation detail. Users wanting to override values in the builtins namespace should import the
The namespace for a module is automatically created the first time a module is imported. The main module for a script is always called __main__.

The `global` statement has the same scope as a name binding operation in the same block. If the nearest enclosing scope for a free variable contains a global statement, the free variable is treated as a global.

A class definition is an executable statement that may use and define names. These references follow the normal rules for name resolution. The namespace of the class definition becomes the attribute dictionary of the class. Names defined at the class scope are not visible in methods.

### 4.1.1. Interaction with dynamic features

There are several cases where Python statements are illegal when used in conjunction with nested scopes that contain free variables.

If a variable is referenced in an enclosing scope, it is illegal to delete the name. An error will be reported at compile time.

If the wild card form of import — `import *` — is used in a function and the function contains or is a nested block with free variables, the compiler will raise a `SyntaxError`.

If `exec` is used in a function and the function contains or is a nested block with free variables, the compiler will raise a `SyntaxError` unless the `exec` explicitly specifies the local namespace for the `exec`. (In other words, `exec obj` would be illegal, but `exec obj in ns` would be
legal.)

The `eval()`, `execfile()`, and `input()` functions and the `exec` statement do not have access to the full environment for resolving names. Names may be resolved in the local and global namespaces of the caller. Free variables are not resolved in the nearest enclosing namespace, but in the global namespace. [1] The `exec` statement and the `eval()` and `execfile()` functions have optional arguments to override the global and local namespace. If only one namespace is specified, it is used for both.
4.2. Exceptions

Exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions. An exception is raised at the point where the error is detected; it may be handled by the surrounding code block or by any code block that directly or indirectly invoked the code block where the error occurred.

The Python interpreter raises an exception when it detects a runtime error (such as division by zero). A Python program can also explicitly raise an exception with the raise statement. Exception handlers are specified with the try ... except statement. The finally clause of such a statement can be used to specify cleanup code which does not handle the exception, but is executed whether an exception occurred or not in the preceding code.

Python uses the “termination” model of error handling: an exception handler can find out what happened and continue execution at an outer level, but it cannot repair the cause of the error and retry the failing operation (except by re-entering the offending piece of code from the top).

When an exception is not handled at all, the interpreter terminates execution of the program, or returns to its interactive main loop. In either case, it prints a stack backtrace, except when the exception is SystemExit.

Exceptions are identified by class instances. The except clause is selected depending on the class of the instance: it must reference the class of the instance or a base class thereof. The instance can be received by the handler and can carry additional information about the exceptional condition.
Exceptions can also be identified by strings, in which case the `except` clause is selected by object identity. An arbitrary value can be raised along with the identifying string which can be passed to the handler.

**Note:** Messages to exceptions are not part of the Python API. Their contents may change from one version of Python to the next without warning and should not be relied on by code which will run under multiple versions of the interpreter.

See also the description of the `try` statement in section The `try` statement and `raise` statement in section The `raise` statement.

**Footnotes**

This limitation occurs because the code that is executed by these operations is not available at the time the module is compiled.

[1]
5. Expressions

This chapter explains the meaning of the elements of expressions in Python.

**Syntax Notes:** In this and the following chapters, extended BNF notation will be used to describe syntax, not lexical analysis. When (one alternative of) a syntax rule has the form

\[
\text{name ::= othername}
\]

and no semantics are given, the semantics of this form of \text{name} are the same as for \text{othername}.
5.1. Arithmetic conversions

When a description of an arithmetic operator below uses the phrase “the numeric arguments are converted to a common type,” the arguments are coerced using the coercion rules listed at Coercion rules. If both arguments are standard numeric types, the following coercions are applied:

- If either argument is a complex number, the other is converted to complex;
- otherwise, if either argument is a floating point number, the other is converted to floating point;
- otherwise, if either argument is a long integer, the other is converted to long integer;
- otherwise, both must be plain integers and no conversion is necessary.

Some additional rules apply for certain operators (e.g., a string left argument to the ‘%’ operator). Extensions can define their own coercions.
5.2. Atoms

Atoms are the most basic elements of expressions. The simplest atoms are identifiers or literals. Forms enclosed in reverse quotes or in parentheses, brackets or braces are also categorized syntactically as atoms. The syntax for atoms is:

\[
\begin{align*}
\text{atom} & ::= \text{identifier} \mid \text{literal} \mid \text{enclosure} \\
\text{enclosure} & ::= \text{parenth_form} \mid \text{list_display} \\
& \quad \mid \text{generator_expression} \mid \text{dict_display} \mid \text{set_display} \\
& \quad \mid \text{string_conversion} \mid \text{yield_atom}
\end{align*}
\]

5.2.1. Identifiers (Names)

An identifier occurring as an atom is a name. See section Identifiers and keywords for lexical definition and section Naming and binding for documentation of naming and binding.

When the name is bound to an object, evaluation of the atom yields that object. When a name is not bound, an attempt to evaluate it raises a NameError exception.

Private name mangling: When an identifier that textually occurs in a class definition begins with two or more underscore characters and does not end in two or more underscores, it is considered a private name of that class. Private names are transformed to a longer form before code is generated for them. The transformation inserts the class name, with leading underscores removed and a single underscore inserted, in front of the name. For example, the identifier \texttt{__spam} occurring in a class named \texttt{Ham} will be transformed to \texttt{_Ham__spam}. This transformation is independent of the syntactical context in which the identifier is used. If the transformed name is extremely long (longer than 255 characters), implementation defined
truncation may happen. If the class name consists only of underscores, no transformation is done.

5.2.2. Literals

Python supports string literals and various numeric literals:

```
literal ::= stringliteral | integer | longinteger | floatnumber | imagnumber
```

Evaluation of a literal yields an object of the given type (string, integer, long integer, floating point number, complex number) with the given value. The value may be approximated in the case of floating point and imaginary (complex) literals. See section Literals for details.

All literals correspond to immutable data types, and hence the object's identity is less important than its value. Multiple evaluations of literals with the same value (either the same occurrence in the program text or a different occurrence) may obtain the same object or a different object with the same value.

5.2.3. Parenthesized forms

A parenthesized form is an optional expression list enclosed in parentheses:

```
parenth_form ::= "( [expression_list] )"
```

A parenthesized expression list yields whatever that expression list yields: if the list contains at least one comma, it yields a tuple; otherwise, it yields the single expression that makes up the expression list.

An empty pair of parentheses yields an empty tuple object. Since
tuples are immutable, the rules for literals apply (i.e., two occurrences of the empty tuple may or may not yield the same object).

Note that tuples are not formed by the parentheses, but rather by use of the comma operator. The exception is the empty tuple, for which parentheses are required — allowing unparenthesized "nothing" in expressions would cause ambiguities and allow common typos to pass uncaught.

5.2.4. List displays

A list display is a possibly empty series of expressions enclosed in square brackets:

```
list_display ::= "[" [expression_list | list_comprehension
list_comprehension ::= expression list_for
list_for ::= "for" target_list "in" old_expression_list
old_expression_list ::= old_expression [""," old_expression]*
old_expression ::= or_test | old_lambda_expr
list_iter ::= list_for | list_if
list_if ::= "if" old_expression [list_iter]
```

A list display yields a new list object. Its contents are specified by providing either a list of expressions or a list comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and placed into the list object in that order. When a list comprehension is supplied, it consists of a single expression followed by at least one for clause and zero or more for or if clauses. In this case, the elements of the new list are those that would be produced by considering each of the for or if clauses a block, nesting from left to right, and evaluating the expression to produce a list element each time the innermost block is reached [1].

5.2.5. Displays for sets and dictionaries
For constructing a set or a dictionary Python provides special syntax called “displays”, each of them in two flavors:

- either the container contents are listed explicitly, or
- they are computed via a set of looping and filtering instructions, called a *comprehension*.

Common syntax elements for comprehensions are:

```plaintext
comprehension ::= expression comp_for
comp_for ::= "for" target_list "in" or_test [comp_iter]
comp_iter ::= comp_for | comp_if
comp_if ::= "if" expression_nocond [comp_iter]
```

The comprehension consists of a single expression followed by at least one *for* clause and zero or more *for* or *if* clauses. In this case, the elements of the new container are those that would be produced by considering each of the *for* or *if* clauses a block, nesting from left to right, and evaluating the expression to produce an element each time the innermost block is reached.

Note that the comprehension is executed in a separate scope, so names assigned to in the target list don't "leak" in the enclosing scope.

### 5.2.6. Generator expressions

A generator expression is a compact generator notation in parentheses:

```plaintext
generator_expression ::= "(" expression comp_for ")"
```

A generator expression yields a new generator object. Its syntax is the same as for comprehensions, except that it is enclosed in parentheses instead of brackets or curly braces.
Variables used in the generator expression are evaluated lazily when the `__next__()` method is called for generator object (in the same fashion as normal generators). However, the leftmost `for` clause is immediately evaluated, so that an error produced by it can be seen before any other possible error in the code that handles the generator expression. Subsequent `for` clauses cannot be evaluated immediately since they may depend on the previous `for` loop. For example: `(x*y for x in range(10) for y in bar(x))`.

The parentheses can be omitted on calls with only one argument. See section Calls for the detail.

### 5.2.7. Dictionary displays

A dictionary display is a possibly empty series of key/datum pairs enclosed in curly braces:

```plaintext
dict_display ::= "{" [key_datum_list | dict_comprehension] "}"
key_datum_list ::= key_datum ("," key_datum)* ["","]
key_datum ::= expression ":" expression
dict_comprehension ::= expression ":" expression comp_for
```

A dictionary display yields a new dictionary object.

If a comma-separated sequence of key/datum pairs is given, they are evaluated from left to right to define the entries of the dictionary: each key object is used as a key into the dictionary to store the corresponding datum. This means that you can specify the same key multiple times in the key/datum list, and the final dictionary's value for that key will be the last one given.

A dict comprehension, in contrast to list and set comprehensions, needs two expressions separated with a colon followed by the usual "for" and "if" clauses. When the comprehension is run, the resulting key and value elements are inserted in the new dictionary in the
order they are produced.

Restrictions on the types of the key values are listed earlier in section The standard type hierarchy. (To summarize, the key type should be hashable, which excludes all mutable objects.) Clashes between duplicate keys are not detected; the last datum (textually rightmost in the display) stored for a given key value prevails.

5.2.8. Set displays

A set display is denoted by curly braces and distinguishable from dictionary displays by the lack of colons separating keys and values:

```
set_display ::= "{" (expression_list | comprehension) "}"`
```

A set display yields a new mutable set object, the contents being specified by either a sequence of expressions or a comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and added to the set object. When a comprehension is supplied, the set is constructed from the elements resulting from the comprehension.

An empty set cannot be constructed with {}; this literal constructs an empty dictionary.

5.2.9. String conversions

A string conversion is an expression list enclosed in reverse (a.k.a. backward) quotes:

```
string_conversion ::= "``" expression_list "``"
```

A string conversion evaluates the contained expression list and converts the resulting object into a string according to rules specific to its type.
If the object is a string, a number,  `None`, or a tuple, list or dictionary containing only objects whose type is one of these, the resulting string is a valid Python expression which can be passed to the built-in function  `eval()`  to yield an expression with the same value (or an approximation, if floating point numbers are involved).

(In particular, converting a string adds quotes around it and converts “funny” characters to escape sequences that are safe to print.)

Recursive objects (for example, lists or dictionaries that contain a reference to themselves, directly or indirectly) use  `...`  to indicate a recursive reference, and the result cannot be passed to  `eval()`  to get an equal value (  `SyntaxError`  will be raised instead).

The built-in function  `repr()`  performs exactly the same conversion in its argument as enclosing it in parentheses and reverse quotes does. The built-in function  `str()`  performs a similar but more user-friendly conversion.

### 5.2.10. Yield expressions

```
yield_atom ::= "(" yield_expression ")"
yield_expression ::= "yield" [expression_list]
```

*New in version 2.5.*

The  `yield` expression is only used when defining a generator function, and can only be used in the body of a function definition. Using a  `yield` expression in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

When a generator function is called, it returns an iterator known as a generator. That generator then controls the execution of a generator function. The execution starts when one of the generator’s methods
is called. At that time, the execution proceeds to the first \texttt{yield}
expression, where it is suspended again, returning the value of
\texttt{expression_list} to generator’s caller. By suspended we mean that
all local state is retained, including the current bindings of local
variables, the instruction pointer, and the internal evaluation stack.
When the execution is resumed by calling one of the generator's
methods, the function can proceed exactly as if the \texttt{yield} expression
was just another external call. The value of the \texttt{yield} expression
after resuming depends on the method which resumed the
execution.

All of this makes generator functions quite similar to coroutines; they
yield multiple times, they have more than one entry point and their
execution can be suspended. The only difference is that a generator
function cannot control where should the execution continue after it
yields; the control is always transferred to the generator’s caller.

\subsection*{5.2.10.1. Generator-iterator methods}

This subsection describes the methods of a generator iterator. They
can be used to control the execution of a generator function.

Note that calling any of the generator methods below when the
generator is already executing raises a \texttt{ValueError} exception.

\texttt{generator.next()}

Starts the execution of a generator function or resumes it at the
last executed \texttt{yield} expression. When a generator function is
resumed with a \texttt{next()} method, the current \texttt{yield} expression
always evaluates to \texttt{None}. The execution then continues to the
next \texttt{yield} expression, where the generator is suspended again,
and the value of the \texttt{expression_list} is returned to \texttt{next()}'s
caller. If the generator exits without yielding another value, a
\texttt{StopIteration} exception is raised.
generator.send(value)

Resumes the execution and “sends” a value into the generator function. The value argument becomes the result of the current yield expression. The send() method returns the next value yielded by the generator, or raises StopIteration if the generator exits without yielding another value. When send() is called to start the generator, it must be called with None as the argument, because there is no yield expression that could receive the value.

generator.throw(type[, value[, traceback]])

Raises an exception of type type at the point where generator was paused, and returns the next value yielded by the generator function. If the generator exits without yielding another value, a StopIteration exception is raised. If the generator function does not catch the passed-in exception, or raises a different exception, then that exception propagates to the caller.

generator.close()

Raises a GeneratorExit at the point where the generator function was paused. If the generator function then raises StopIteration (by exiting normally, or due to already being closed) or GeneratorExit (by not catching the exception), close returns to its caller. If the generator yields a value, a RuntimeError is raised. If the generator raises any other exception, it is propagated to the caller. close() does nothing if the generator has already exited due to an exception or normal exit.

Here is a simple example that demonstrates the behavior of generators and generator functions:

```python
>>> def echo(value=None):
...     print "Execution starts when 'next()' is called for the first time."
...     try:
```
```python
... while True:
    try:
    ...
        value = (yield value)
    ...
    except Exception, e:
        ...
            value = e
    ...
    finally:
    ...
        print "Don't forget to clean up when 'close()' is called." ... 

>>> generator = echo(1)
>>> print generator.next()
Execution starts when 'next()' is called for the first time.
1
>>> print generator.next()
None
>>> print generator.send(2)
2
>>> generator.throw(TypeError, "spam")
TypeError('spam',)
>>> generator.close()
Don't forget to clean up when 'close()' is called.
```

See also:

**PEP 342 - Coroutines via Enhanced Generators**

The proposal to enhance the API and syntax of generators, making them usable as simple coroutines.
5.3. Primaries

Primaries represent the most tightly bound operations of the language. Their syntax is:

```
primary ::= atom | attributeref | subscription | slicing | call
```

5.3.1. Attribute references

An attribute reference is a primary followed by a period and a name:

```
attributeref ::= primary "." identifier
```

The primary must evaluate to an object of a type that supports attribute references, e.g., a module, list, or an instance. This object is then asked to produce the attribute whose name is the identifier. If this attribute is not available, the exception `AttributeError` is raised. Otherwise, the type and value of the object produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

5.3.2. Subscriptions

A subscription selects an item of a sequence (string, tuple or list) or mapping (dictionary) object:

```
subscription ::= primary "[" expression_list "]"
```

The primary must evaluate to an object of a sequence or mapping type.

If the primary is a mapping, the expression list must evaluate to an object whose value is one of the keys of the mapping, and the
subscription selects the value in the mapping that corresponds to that key. (The expression list is a tuple except if it has exactly one item.)

If the primary is a sequence, the expression (list) must evaluate to a plain integer. If this value is negative, the length of the sequence is added to it (so that, e.g., \( x[-1] \) selects the last item of \( x \).) The resulting value must be a nonnegative integer less than the number of items in the sequence, and the subscription selects the item whose index is that value (counting from zero).

A string's items are characters. A character is not a separate data type but a string of exactly one character.

### 5.3.3. Slicings

A slicing selects a range of items in a sequence object (e.g., a string, tuple or list). Slicings may be used as expressions or as targets in assignment or `del` statements. The syntax for a slicing:

```
slicing ::= simple_slicing | extended_slicing
simple_slicing ::= primary "[" short_slice "]"
extended_slicing ::= primary "[" slice_list "]"
slice_list ::= slice_item ("," slice_item)* [",",]
slice_item ::= expression | proper_slice | ellipsis
proper_slice ::= short_slice | long_slice
short_slice ::= [lower_bound] ":" [upper_bound]
long_slice ::= short_slice ":" [stride]
lower_bound ::= expression
upper_bound ::= expression
stride ::= expression
ellipsis ::= "..."
```

There is ambiguity in the formal syntax here: anything that looks like an expression list also looks like a slice list, so any subscription can be interpreted as a slicing. Rather than further complicating the syntax, this is disambiguated by defining that in this case the interpretation as a subscription takes priority over the interpretation...
as a slicing (this is the case if the slice list contains no proper slice nor ellipses). Similarly, when the slice list has exactly one short slice and no trailing comma, the interpretation as a simple slicing takes priority over that as an extended slicing.

The semantics for a simple slicing are as follows. The primary must evaluate to a sequence object. The lower and upper bound expressions, if present, must evaluate to plain integers; defaults are zero and the \texttt{sys.maxint}, respectively. If either bound is negative, the sequence's length is added to it. The slicing now selects all items with index \( k \) such that \( i \leq k < j \) where \( i \) and \( j \) are the specified lower and upper bounds. This may be an empty sequence. It is not an error if \( i \) or \( j \) lie outside the range of valid indexes (such items don't exist so they aren't selected).

The semantics for an extended slicing are as follows. The primary must evaluate to a mapping object, and it is indexed with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of an ellipsis slice item is the built-in \texttt{Ellipsis} object. The conversion of a proper slice is a slice object (see section \textbf{The standard type hierarchy}) whose \texttt{start}, \texttt{stop} and \texttt{step} attributes are the values of the expressions given as lower bound, upper bound and stride, respectively, substituting \texttt{None} for missing expressions.

### 5.3.4. Calls

A call calls a callable object (e.g., a \texttt{function}) with a possibly empty series of arguments:

\[
\text{call} ::= \text{primary } "(" \ [\text{argument_list } [",",] \]
\]
A trailing comma may be present after the positional and keyword arguments but does not affect the semantics.

The primary must evaluate to a callable object (user-defined functions, built-in functions, methods of built-in objects, class objects, methods of class instances, and certain class instances themselves are callable; extensions may define additional callable object types). All argument expressions are evaluated before the call is attempted. Please refer to section Function definitions for the syntax of formal parameter lists.

If keyword arguments are present, they are first converted to positional arguments, as follows. First, a list of unfilled slots is created for the formal parameters. If there are N positional arguments, they are placed in the first N slots. Next, for each keyword argument, the identifier is used to determine the corresponding slot (if the identifier is the same as the first formal parameter name, the first slot is used, and so on). If the slot is already filled, a TypeError exception is raised. Otherwise, the value of the argument is placed in the slot, filling it (even if the expression is None, it fills the slot). When all arguments have been processed, the slots that are still unfilled are filled with the corresponding default value from the function definition. (Default values are calculated, once, when the function is defined; thus, a mutable object such as a list or dictionary used as default value will be shared by all calls that
don’t specify an argument value for the corresponding slot; this should usually be avoided.) If there are any unfilled slots for which no default value is specified, a `TypeError` exception is raised. Otherwise, the list of filled slots is used as the argument list for the call.

**CPython implementation detail:** An implementation may provide built-in functions whose positional parameters do not have names, even if they are ‘named’ for the purpose of documentation, and which therefore cannot be supplied by keyword. In CPython, this is the case for functions implemented in C that use `PyArg_ParseTuple()` to parse their arguments.

If there are more positional arguments than there are formal parameter slots, a `TypeError` exception is raised, unless a formal parameter using the syntax `*identifier` is present; in this case, that formal parameter receives a tuple containing the excess positional arguments (or an empty tuple if there were no excess positional arguments).

If any keyword argument does not correspond to a formal parameter name, a `TypeError` exception is raised, unless a formal parameter using the syntax `**identifier` is present; in this case, that formal parameter receives a dictionary containing the excess keyword arguments (using the keywords as keys and the argument values as corresponding values), or a (new) empty dictionary if there were no excess keyword arguments.

If the syntax `*expression` appears in the function call, `expression` must evaluate to an iterable. Elements from this iterable are treated as if they were additional positional arguments; if there are positional arguments `x1, ..., xN`, and `expression` evaluates to a sequence `y1, 
..., $y_M$, this is equivalent to a call with $M+N$ positional arguments $x_1$, ..., $x_N$, $y_1$, ..., $y_M$.

A consequence of this is that although the *expression syntax may appear after some keyword arguments, it is processed before the keyword arguments (and the **expression argument, if any – see below). So:

```
>>> def f(a, b):
...   print a, b
... >>> f(b=1, *(2,))
2 1
>>> f(a=1, *(2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1, *(2,))
1 2
```

It is unusual for both keyword arguments and the *expression syntax to be used in the same call, so in practice this confusion does not arise.

If the syntax **expression appears in the function call, expression must evaluate to a mapping, the contents of which are treated as additional keyword arguments. In the case of a keyword appearing in both expression and as an explicit keyword argument, a TypeError exception is raised.

Formal parameters using the syntax *identifier or **identifier cannot be used as positional argument slots or as keyword argument names. Formal parameters using the syntax (sublist) cannot be used as keyword argument names; the outermost sublist corresponds to a single unnamed argument slot, and the argument value is assigned to the sublist using the usual tuple assignment rules after all other parameter processing is done.
A call always returns some value, possibly None, unless it raises an exception. How this value is computed depends on the type of the callable object.

If it is—

a user-defined function:
   The code block for the function is executed, passing it the argument list. The first thing the code block will do is bind the formal parameters to the arguments; this is described in section Function definitions. When the code block executes a return statement, this specifies the return value of the function call.

a built-in function or method:
   The result is up to the interpreter; see Built-in Functions for the descriptions of built-in functions and methods.

a class object:
   A new instance of that class is returned.

a class instance method:
   The corresponding user-defined function is called, with an argument list that is one longer than the argument list of the call: the instance becomes the first argument.

a class instance:
   The class must define a __call__() method; the effect is then the same as if that method was called.
5.4. The power operator

The power operator binds more tightly than unary operators on its left; it binds less tightly than unary operators on its right. The syntax is:

\[
\text{power ::= primary } ["**" \text{ u_expr}]
\]

Thus, in an unparenthesized sequence of power and unary operators, the operators are evaluated from right to left (this does not constrain the evaluation order for the operands): \(-1**2\) results in \(-1\).

The power operator has the same semantics as the built-in `pow()` function, when called with two arguments: it yields its left argument raised to the power of its right argument. The numeric arguments are first converted to a common type. The result type is that of the arguments after coercion.

With mixed operand types, the coercion rules for binary arithmetic operators apply. For int and long int operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, \(10**2\) returns 100, but \(10**-2\) returns 0.01. (This last feature was added in Python 2.2. In Python 2.1 and before, if both arguments were of integer types and the second argument was negative, an exception was raised).

Raising 0.0 to a negative power results in a `ZeroDivisionError`. Raising a negative number to a fractional power results in a `ValueError`.
5.5. Unary arithmetic and bitwise operations

All unary arithmetic and bitwise operations have the same priority:

```
u_expr ::= power | "-" u_expr | "+" u_expr | "~" u_expr
```

The unary `-` (minus) operator yields the negation of its numeric argument.

The unary `+` (plus) operator yields its numeric argument unchanged.

The unary `~` (invert) operator yields the bitwise inversion of its plain or long integer argument. The bitwise inversion of $x$ is defined as $-(x+1)$. It only applies to integral numbers.

In all three cases, if the argument does not have the proper type, a `TypeError` exception is raised.
5.6. Binary arithmetic operations

The binary arithmetic operations have the conventional priority levels. Note that some of these operations also apply to certain non-numeric types. Apart from the power operator, there are only two levels, one for multiplicative operators and one for additive operators:

```
m_expr ::= u_expr | m_expr "*" u_expr | m_expr "/" u_expr |
            | m_expr "%" u_expr
a_expr ::= m_expr | a_expr "+" m_expr | a_expr "-" m_expr
```

The * (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, or one argument must be an integer (plain or long) and the other must be a sequence. In the former case, the numbers are converted to a common type and then multiplied together. In the latter case, sequence repetition is performed; a negative repetition factor yields an empty sequence.

The / (division) and // (floor division) operators yield the quotient of their arguments. The numeric arguments are first converted to a common type. Plain or long integer division yields an integer of the same type; the result is that of mathematical division with the ‘floor’ function applied to the result. Division by zero raises the `ZeroDivisionError` exception.

The % (modulo) operator yields the remainder from the division of the first argument by the second. The numeric arguments are first converted to a common type. A zero right argument raises the `ZeroDivisionError` exception. The arguments may be floating point numbers, e.g., 3.14%0.7 equals 0.34 (since 3.14 equals 4*0.7 + 0.34.) The modulo operator always yields a result with the same sign as its second operand (or zero); the absolute value of the result is
strictly smaller than the absolute value of the second operand [2].

The integer division and modulo operators are connected by the following identity: $x \equiv (x/y) \times y + (x \bmod y)$. Integer division and modulo are also connected with the built-in function `divmod()`: `divmod(x, y) == (x/y, x \bmod y)`. These identities don’t hold for floating point numbers; there similar identities hold approximately where $x/y$ is replaced by `floor(x/y)` or `floor(x/y) - 1` [3].

In addition to performing the modulo operation on numbers, the `%` operator is also overloaded by string and unicode objects to perform string formatting (also known as interpolation). The syntax for string formatting is described in the Python Library Reference, section String Formatting Operations.

**Deprecated since version 2.3:** The floor division operator, the modulo operator, and the `divmod()` function are no longer defined for complex numbers. Instead, convert to a floating point number using the `abs()` function if appropriate.

The `+` (addition) operator yields the sum of its arguments. The arguments must either both be numbers or both sequences of the same type. In the former case, the numbers are converted to a common type and then added together. In the latter case, the sequences are concatenated.

The `-` (subtraction) operator yields the difference of its arguments. The numeric arguments are first converted to a common type.
5.7. Shifting operations

The shifting operations have lower priority than the arithmetic operations:

```
shift_expr ::= a_expr | shift_expr ( "<<" | ">>" ) a_expr
```

These operators accept plain or long integers as arguments. The arguments are converted to a common type. They shift the first argument to the left or right by the number of bits given by the second argument.

A right shift by \( n \) bits is defined as division by \( \text{pow}(2, n) \). A left shift by \( n \) bits is defined as multiplication with \( \text{pow}(2, n) \). Negative shift counts raise a `ValueError` exception.

**Note:** In the current implementation, the right-hand operand is required to be at most `sys.maxsize`. If the right-hand operand is larger than `sys.maxsize` an `OverflowError` exception is raised.
5.8. Binary bitwise operations

Each of the three bitwise operations has a different priority level:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`and_expr ::= shift_expr</td>
<td>and_expr &quot;&amp;&quot; shift_expr`</td>
</tr>
<tr>
<td>`xor_expr ::= and_expr</td>
<td>xor_expr &quot;^&quot; and_expr`</td>
</tr>
<tr>
<td>`or_expr ::= xor_expr</td>
<td>or_expr &quot;</td>
</tr>
</tbody>
</table>
5.9. Comparisons

Unlike C, all comparison operations in Python have the same priority, which is lower than that of any arithmetic, shifting or bitwise operation. Also unlike C, expressions like \( a < b < c \) have the interpretation that is conventional in mathematics:

\[
\text{comparison ::= or_expr ( comp_operator or_expr )}^*
\]

\[
\text{comp_operator ::= } "<" \mid ">" \mid "==" \mid ">=" \mid "<=" \mid "<>" \mid ">=" \mid ">=" \mid ">=" \mid "is" \mid "not" \mid "not" \text{ "in"}
\]

Comparisons yield boolean values: True or False.

Comparisons can be chained arbitrarily, e.g., \( x < y \leq z \) is equivalent to \( x < y \) and \( y \leq z \), except that \( y \) is evaluated only once (but in both cases \( z \) is not evaluated at all when \( x < y \) is found to be false).

Formally, if \( a, b, c, \ldots, y, z \) are expressions and \( op1, op2, \ldots, opN \) are comparison operators, then \( a \ op1 \ b \ op2 \ c \ldots \ y \ opN \ z \) is equivalent to \( a \ op1 \ b \) and \( b \ op2 \ c \) and \( \ldots \) \( y \ opN \ z \), except that each expression is evaluated at most once.

Note that \( a \ op1 \ b \ op2 \ c \) doesn’t imply any kind of comparison between \( a \) and \( c \), so that, e.g., \( x < y > z \) is perfectly legal (though perhaps not pretty).

The forms \( <> \) and \( != \) are equivalent; for consistency with C, \( != \) is preferred; where \( != \) is mentioned below \( <> \) is also accepted. The \( <> \) spelling is considered obsolescent.

The operators \( <, >, ==, >=, <=, != \) compare the values of two objects. The objects need not have the same type. If both are
numbers, they are converted to a common type. Otherwise, objects of different types *always* compare unequal, and are ordered consistently but arbitrarily. You can control comparison behavior of objects of non-built-in types by defining a \_\_cmp\_\_ method or rich comparison methods like \_\_gt\_\_, described in section *Special method names*.

(This unusual definition of comparison was used to simplify the definition of operations like sorting and the *in* and *not in* operators. In the future, the comparison rules for objects of different types are likely to change.)

Comparison of objects of the same type depends on the type:

- Numbers are compared arithmetically.
- Strings are compared lexicographically using the numeric equivalents (the result of the built-in function \*ord\*(\*)) of their characters. Unicode and 8-bit strings are fully interoperable in this behavior. [4]
- Tuples and lists are compared lexicographically using comparison of corresponding elements. This means that to compare equal, each element must compare equal and the two sequences must be of the same type and have the same length.

If not equal, the sequences are ordered the same as their first differing elements. For example, \*cmp\*(([1,2,x]), [1,2,y]) returns the same as \*cmp\*(x,y). If the corresponding element does not exist, the shorter sequence is ordered first (for example, [1,2] < [1,2,3]).

- Mappings (dictionaries) compare equal if and only if their sorted (key, value) lists compare equal. [5] Outcomes other than equality are resolved consistently, but are not otherwise defined.
Most other objects of built-in types compare unequal unless they are the same object; the choice whether one object is considered smaller or larger than another one is made arbitrarily but consistently within one execution of a program.

The operators in and not in test for collection membership. \( x \text{ in } s \) evaluates to true if \( x \) is a member of the collection \( s \), and false otherwise. \( x \text{ not in } s \) returns the negation of \( x \text{ in } s \). The collection membership test has traditionally been bound to sequences; an object is a member of a collection if the collection is a sequence and contains an element equal to that object. However, it make sense for many other object types to support membership tests without being a sequence. In particular, dictionaries (for keys) and sets support membership testing.

For the list and tuple types, \( x \text{ in } y \) is true if and only if there exists an index \( i \) such that either \( x \text{ is } y[i] \) or \( x == y[i] \) is true.

For the Unicode and string types, \( x \text{ in } y \) is true if and only if \( x \) is a substring of \( y \). An equivalent test is \( y.find(x) \neq -1 \). Note, \( x \) and \( y \) need not be the same type; consequently, \( u'ab' \text{ in } 'abc' \) will return True. Empty strings are always considered to be a substring of any other string, so "" in "abc" will return True.

*Changed in version 2.3:* Previously, \( x \) was required to be a string of length 1.

For user-defined classes which define the __contains__() method, \( x \text{ in } y \) is true if and only if \( y.__contains__(x) \) is true.

For user-defined classes which do not define __contains__() but do define __iter__(), \( x \text{ in } y \) is true if some value \( z \) with \( x == z \) is
produced while iterating over \( y \). If an exception is raised during the iteration, it is as if \texttt{in} raised that exception.

Lastly, the old-style iteration protocol is tried: if a class defines \texttt{__getitem__()}, \texttt{x in y} is true if and only if there is a non-negative integer index \( i \) such that \( x == y[i] \), and all lower integer indices do not raise \texttt{IndexError} exception. (If any other exception is raised, it is as if \texttt{in} raised that exception).

The operator \texttt{not in} is defined to have the inverse true value of \texttt{in}.

The operators \texttt{is} and \texttt{is not} test for object identity: \texttt{x is y} is true if and only if \( x \) and \( y \) are the same object. \texttt{x is not y} yields the inverse truth value. [7]
5.10. Boolean operations

In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: `False`, `None`, numeric zero of all types, and empty strings and containers (including strings, tuples, lists, dictionaries, sets and frozensets). All other values are interpreted as true. (See the `__nonzero__()` special method for a way to change this.)

The operator `not` yields `True` if its argument is false, `False` otherwise.

The expression `x and y` first evaluates `x`; if `x` is false, its value is returned; otherwise, `y` is evaluated and the resulting value is returned.

The expression `x or y` first evaluates `x`; if `x` is true, its value is returned; otherwise, `y` is evaluated and the resulting value is returned.

(Note that neither `and` nor `or` restrict the value and type they return to `False` and `True`, but rather return the last evaluated argument. This is sometimes useful, e.g., if `s` is a string that should be replaced by a default value if it is empty, the expression `s or 'foo'` yields the desired value. Because `not` has to invent a value anyway, it does not bother to return a value of the same type as its argument, so e.g., `not 'foo'` yields `False`, not `''`.)
5.11. Conditional Expressions

New in version 2.5.

```
conditional_expression ::= or_test ["if" or_test "else" expression
expression ::= conditional_expression | lambda_expr
```

Conditional expressions (sometimes called a “ternary operator”) have the lowest priority of all Python operations.

The expression `x if C else y` first evaluates the condition, `C (not x)`; if `C` is true, `x` is evaluated and its value is returned; otherwise, `y` is evaluated and its value is returned.

See PEP 308 for more details about conditional expressions.
5.12. Lambdas

<table>
<thead>
<tr>
<th><code>lambda_expr</code></th>
<th>::=</th>
<th>&quot;lambda&quot; [parameter_list]: expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>old_lambda_expr</code></td>
<td>::=</td>
<td>&quot;lambda&quot; [parameter_list]: old_expression</td>
</tr>
</tbody>
</table>

Lambda expressions (sometimes called lambda forms) have the same syntactic position as expressions. They are a shorthand to create anonymous functions; the expression `lambda arguments: expression` yields a function object. The unnamed object behaves like a function object defined with

```python
def name(arguments):
    return expression
```

See section Function definitions for the syntax of parameter lists. Note that functions created with lambda expressions cannot contain statements.
An expression list containing at least one comma yields a tuple. The length of the tuple is the number of expressions in the list. The expressions are evaluated from left to right.

The trailing comma is required only to create a single tuple (a.k.a. a singleton); it is optional in all other cases. A single expression without a trailing comma doesn’t create a tuple, but rather yields the value of that expression. (To create an empty tuple, use an empty pair of parentheses: ()..)
5.14. Evaluation order

Python evaluates expressions from left to right. Notice that while evaluating an assignment, the right-hand side is evaluated before the left-hand side.

In the following lines, expressions will be evaluated in the arithmetic order of their suffixes:

```python
eexpr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
eexpr1 + expr2 * (expr3 - expr4)
eexpr1(expr2, expr3, *expr4, **expr5)
eexpr3, expr4 = expr1, expr2
```
The following table summarizes the operator precedences in Python, from lowest precedence (least binding) to highest precedence (most binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for comparisons, including tests, which all have the same precedence and chain from left to right — see section *Comparisons* — and exponentiation, which groups from right to left).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lambda</td>
<td>Lambda expression</td>
</tr>
<tr>
<td>if – else</td>
<td>Conditional expression</td>
</tr>
<tr>
<td>or</td>
<td>Boolean OR</td>
</tr>
<tr>
<td>and</td>
<td>Boolean AND</td>
</tr>
<tr>
<td>not x</td>
<td>Boolean NOT</td>
</tr>
<tr>
<td>in, not in, is, is not, &lt;, &lt;=, &gt;, &gt;=, &lt;&gt;!, !=, ==</td>
<td>Comparisons, including membership tests and identity tests</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>x[index], x[index:index], x(arguments...), x.attribute</td>
<td>Subscription, slicing, call, attribute reference</td>
</tr>
</tbody>
</table>
In Python 2.3 and later releases, a list comprehension "leaks" the control variables of each for it contains into the containing scope. However, this behavior is deprecated, and relying on it will not work in Python 3.

While \( \text{abs}(x \% y) < \text{abs}(y) \) is true mathematically, for floats it may not be true numerically due to roundoff. For example, and assuming a platform on which a Python float is an IEEE 754 double-precision number, in order that \(-1e-100 \% 1e100\) have the same sign as \(1e100\), the computed result is \(-1e-100 + 1e100\), which is numerically exactly equal to \(1e100\). The function \texttt{math.fmod()} returns a result whose sign matches the sign of the first argument instead, and so returns \(-1e-100\) in this case. Which approach is more appropriate depends on the application.

If \(x\) is very close to an exact integer multiple of \(y\), it's possible for \(\text{floor}(x/y)\) to be one larger than \((x-x\%y)/y\) due to rounding. In such cases, Python returns the latter result, in order to preserve that \(\text{divmod}(x,y)[0] \times y + x \% y\) be very close to \(x\).

While comparisons between unicode strings make sense at the byte level, they may be counter-intuitive to users. For example, the strings \u"\u00C7"\ and \u"\u0043\u0327"\ compare differently, even though they both represent the same unicode character (LATIN CAPITAL LETTER C WITH CEDILLA). To compare strings in a human recognizable way, compare using \texttt{unicodedata.normalize()}. The implementation computes this efficiently, without constructing lists or sorting.

Earlier versions of Python used lexicographic comparison of the sorted (key, value) lists, but this was very expensive for the common case of comparing for equality. An even earlier version of Python compared dictionaries by identity only, but this
caused surprises because people expected to be able to test a dictionary for emptiness by comparing it to `{}`.

Due to automatic garbage-collection, free lists, and the dynamic nature of descriptors, you may notice seemingly unusual behaviour in certain uses of the `is` operator, like those involving comparisons between instance methods, or constants. Check their documentation for more info.

The `%` operator is also used for string formatting; the same precedence applies.

The power operator `**` binds less tightly than an arithmetic or bitwise unary operator on its right, that is, `2**-1` is `0.5`. 

---

[7] Due to automatic garbage-collection, free lists, and the dynamic nature of descriptors, you may notice seemingly unusual behaviour in certain uses of the `is` operator, like those involving comparisons between instance methods, or constants. Check their documentation for more info.

[8] The `%` operator is also used for string formatting; the same precedence applies.

[9] The power operator `**` binds less tightly than an arithmetic or bitwise unary operator on its right, that is, `2**-1` is `0.5`. 

6. Simple statements

Simple statements are comprised within a single logical line. Several simple statements may occur on a single line separated by semicolons. The syntax for simple statements is:

```
simple_stmt ::= expression_stmt |
                assert_stmt |
                assignment_stmt |
                augmented_assignment_stmt |
                pass_stmt |
                del_stmt |
                print_stmt |
                return_stmt |
                yield_stmt |
                raise_stmt |
                break_stmt |
                continue_stmt |
                import_stmt |
                global_stmt |
                exec_stmt
```
6.1. Expression statements

Expression statements are used (mostly interactively) to compute and write a value, or (usually) to call a procedure (a function that returns no meaningful result; in Python, procedures return the value None). Other uses of expression statements are allowed and occasionally useful. The syntax for an expression statement is:

```
expression_stmt ::= expression_list
```

An expression statement evaluates the expression list (which may be a single expression).

In interactive mode, if the value is not None, it is converted to a string using the built-in repr() function and the resulting string is written to standard output (see section The print statement) on a line by itself. (Expression statements yielding None are not written, so that procedure calls do not cause any output.)
6.2. Assignment statements

Assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects:

\[
\text{assignment_stmt} ::= (\text{target_list} "=")\+ (\text{expression_list} \mid \text{yield_expression})
\]

\[
\text{target_list} ::= \text{target} (\"\", \text{target})\* [\"\",\]
\]

\[
\text{target} ::= \text{identifier} \mid \"(\text{target_list })\" \mid \"[\text{target_list }]\" \mid \text{attributeref} \mid \text{subscription} \mid \text{slicing}
\]

(See section Primaries for the syntax definitions for the last three symbols.)

An assignment statement evaluates the expression list (remember that this can be a single expression or a comma-separated list, the latter yielding a tuple) and assigns the single resulting object to each of the target lists, from left to right.

Assignment is defined recursively depending on the form of the target (list). When a target is part of a mutable object (an attribute reference, subscription or slicing), the mutable object must ultimately perform the assignment and decide about its validity, and may raise an exception if the assignment is unacceptable. The rules observed by various types and the exceptions raised are given with the definition of the object types (see section The standard type hierarchy).

Assignment of an object to a target list is recursively defined as follows.

- If the target list is a single target: The object is assigned to that
target.

- If the target list is a comma-separated list of targets: The object must be an iterable with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets.

Assignment of an object to a single target is recursively defined as follows.

- If the target is an identifier (name):
  
  - If the name does not occur in a `global` statement in the current code block: the name is bound to the object in the current local namespace.
  
  - Otherwise: the name is bound to the object in the current global namespace.

The name is rebound if it was already bound. This may cause the reference count for the object previously bound to the name to reach zero, causing the object to be deallocated and its destructor (if it has one) to be called.

- If the target is a target list enclosed in parentheses or in square brackets: The object must be an iterable with the same number of items as there are targets in the target list, and its items are assigned, from left to right, to the corresponding targets.

- If the target is an attribute reference: The primary expression in the reference is evaluated. It should yield an object with assignable attributes; if this is not the case, `TypeError` is raised. That object is then asked to assign the assigned object to the given attribute; if it cannot perform the assignment, it raises an exception (usually but not necessarily `AttributeError`).

Note: If the object is a class instance and the attribute reference
occurs on both sides of the assignment operator, the RHS expression, \( a.x \) can access either an instance attribute or (if no instance attribute exists) a class attribute. The LHS target \( a.x \) is always set as an instance attribute, creating it if necessary. Thus, the two occurrences of \( a.x \) do not necessarily refer to the same attribute: if the RHS expression refers to a class attribute, the LHS creates a new instance attribute as the target of the assignment:

```python
class Cls:
    x = 3           # class variable
inst = Cls()
inst.x = inst.x + 1  # writes inst.x as 4 leaving Cls.x as
```

This description does not necessarily apply to descriptor attributes, such as properties created with `property()`.

- If the target is a subscription: The primary expression in the reference is evaluated. It should yield either a mutable sequence object (such as a list) or a mapping object (such as a dictionary). Next, the subscript expression is evaluated.

If the primary is a mutable sequence object (such as a list), the subscript must yield a plain integer. If it is negative, the sequence's length is added to it. The resulting value must be a nonnegative integer less than the sequence's length, and the sequence is asked to assign the assigned object to its item with that index. If the index is out of range, `IndexError` is raised (assignment to a subscripted sequence cannot add new items to a list).

If the primary is a mapping object (such as a dictionary), the subscript must have a type compatible with the mapping's key type, and the mapping is then asked to create a key/datum pair which maps the subscript to the assigned object. This can either
replace an existing key/value pair with the same key value, or insert a new key/value pair (if no key with the same value existed).

- If the target is a slicing: The primary expression in the reference is evaluated. It should yield a mutable sequence object (such as a list). The assigned object should be a sequence object of the same type. Next, the lower and upper bound expressions are evaluated, insofar they are present; defaults are zero and the sequence’s length. The bounds should evaluate to (small) integers. If either bound is negative, the sequence's length is added to it. The resulting bounds are clipped to lie between zero and the sequence’s length, inclusive. Finally, the sequence object is asked to replace the slice with the items of the assigned sequence. The length of the slice may be different from the length of the assigned sequence, thus changing the length of the target sequence, if the object allows it.

**CPython implementation detail:** In the current implementation, the syntax for targets is taken to be the same as for expressions, and invalid syntax is rejected during the code generation phase, causing less detailed error messages.

**WARNING:** Although the definition of assignment implies that overlaps between the left-hand side and the right-hand side are ‘safe’ (for example \texttt{a, b = b, a} swaps two variables), overlaps \textit{within} the collection of assigned-to variables are not safe! For instance, the following program prints \texttt{[0, 2]}:

```python
x = [0, 1]
i = 0
i, x[i] = 1, 2
print x
```
### 6.2.1. Augmented assignment statements

Augmented assignment is the combination, in a single statement, of a binary operation and an assignment statement:

```
augmented_assignment_stmt ::= augtarget augop (expression_list | augtarget)
augtarget ::= identifier | attributeref | subscription
augop ::= "+=" | "-=" | "+=" | "/=" | "/=/
| "&=" | "^=" | "|=
```

(See section [Primaries](#) for the syntax definitions for the last three symbols.)

An augmented assignment evaluates the target (which, unlike normal assignment statements, cannot be an unpacking) and the expression list, performs the binary operation specific to the type of assignment on the two operands, and assigns the result to the original target. The target is only evaluated once.

An augmented assignment expression like `x += 1` can be rewritten as `x = x + 1` to achieve a similar, but not exactly equal effect. In the augmented version, `x` is only evaluated once. Also, when possible, the actual operation is performed *in-place*, meaning that rather than creating a new object and assigning that to the target, the old object is modified instead.

With the exception of assigning to tuples and multiple targets in a single statement, the assignment done by augmented assignment statements is handled the same way as normal assignments. Similarly, with the exception of the possible *in-place* behavior, the binary operation performed by augmented assignment is the same as the normal binary operations.

For targets which are attribute references, the same [caveat about class and instance attributes](#) applies as for regular assignments.
6.3. The assert statement

Assert statements are a convenient way to insert debugging assertions into a program:

```
assert_stmt ::= "assert" expression ["," expression]
```

The simple form, `assert expression`, is equivalent to

```
if __debug__:  
    if not expression: raise AssertionError
```

The extended form, `assert expression1, expression2`, is equivalent to

```
if __debug__:  
    if not expression1: raise AssertionError(expression2)
```

These equivalences assume that `__debug__` and `AssertionError` refer to the built-in variables with those names. In the current implementation, the built-in variable `__debug__` is `True` under normal circumstances, `False` when optimization is requested (command line option `-O`). The current code generator emits no code for an assert statement when optimization is requested at compile time. Note that it is unnecessary to include the source code for the expression that failed in the error message; it will be displayed as part of the stack trace.

Assignments to `__debug__` are illegal. The value for the built-in variable is determined when the interpreter starts.
6.4. The pass statement

\[
pass\text{\_stmt} ::= \text{"pass"}
\]

\textit{pass} is a null operation — when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed, for example:

\begin{verbatim}
def f(arg): pass  # a function that does nothing (yet)
class C: pass    # a class with no methods (yet)
\end{verbatim}
6.5. The `del` statement

```
\texttt{delStmt ::= "del" targetList}
```

Deletion is recursively defined very similar to the way assignment is defined. Rather than spelling it out in full details, here are some hints.

Deletion of a target list recursively deletes each target, from left to right.

Deletion of a name removes the binding of that name from the local or global namespace, depending on whether the name occurs in a `global` statement in the same code block. If the name is unbound, a `NameError` exception will be raised.

It is illegal to delete a name from the local namespace if it occurs as a free variable in a nested block.

Deletion of attribute references, subscriptions and slicings is passed to the primary object involved; deletion of a slicing is in general equivalent to assignment of an empty slice of the right type (but even this is determined by the sliced object).
print statement

print evaluates each expression in turn and writes the resulting object to standard output (see below). If an object is not a string, it is first converted to a string using the rules for string conversions. The (resulting or original) string is then written. A space is written before each object is (converted and) written, unless the output system believes it is positioned at the beginning of a line. This is the case (1) when no characters have yet been written to standard output, (2) when the last character written to standard output is a whitespace character except ' ', or (3) when the last write operation on standard output was not a print statement. (In some cases it may be functional to write an empty string to standard output for this reason.)

Note: Objects which act like file objects but which are not the built-in file objects often do not properly emulate this aspect of the file object's behavior, so it is best not to rely on this.

A ' \n ' character is written at the end, unless the print statement ends with a comma. This is the only action if the statement contains just the keyword print.

Standard output is defined as the file object named stdout in the built-in module sys. If no such object exists, or if it does not have a write() method, a RuntimeError exception is raised.

print also has an extended form, defined by the second portion of the syntax described above. This form is sometimes referred to as
“print chevron.” In this form, the first expression after the >> must evaluate to a “file-like” object, specifically an object that has a write() method as described above. With this extended form, the subsequent expressions are printed to this file object. If the first expression evaluates to None, then sys.stdout is used as the file for output.
6.7. The `return` statement

```
return_stmt ::= "return" [expression_list]
```

`return` may only occur syntactically nested in a function definition, not within a nested class definition.

If an expression list is present, it is evaluated, else `None` is substituted.

`return` leaves the current function call with the expression list (or `None`) as return value.

When `return` passes control out of a `try` statement with a `finally` clause, that `finally` clause is executed before really leaving the function.

In a generator function, the `return` statement is not allowed to include an `expression_list`. In that context, a bare `return` indicates that the generator is done and will cause `StopIteration` to be raised.
6.8. The `yield` statement

| yield_stmt ::= yield_expression |

The `yield` statement is only used when defining a generator function, and is only used in the body of the generator function. Using a `yield` statement in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

When a generator function is called, it returns an iterator known as a generator iterator, or more commonly, a generator. The body of the generator function is executed by calling the generator’s `next()` method repeatedly until it raises an exception.

When a `yield` statement is executed, the state of the generator is frozen and the value of `expression_list` is returned to `next()`'s caller. By “frozen” we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, and the internal evaluation stack: enough information is saved so that the next time `next()` is invoked, the function can proceed exactly as if the `yield` statement were just another external call.

As of Python version 2.5, the `yield` statement is now allowed in the `try` clause of a `try ... finally` construct. If the generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), the generator-iterator’s `close()` method will be called, allowing any pending `finally` clauses to execute.

For full details of `yield` semantics, refer to the `Yield expressions` section.
Note: In Python 2.2, the `yield` statement was only allowed when the `generators` feature has been enabled. This `__future__` import statement was used to enable the feature:

```python
from __future__ import generators
```

See also:

**PEP 255 - Simple Generators**

The proposal for adding generators and the `yield` statement to Python.

**PEP 342 - Coroutines via Enhanced Generators**

The proposal that, among other generator enhancements, proposed allowing `yield` to appear inside a `try ... finally` block.
6.9. The \texttt{raise} statement

\begin{verbatim}
raise_stmt ::= "raise" [expression ["," expression ["," expression]]]
\end{verbatim}

If no expressions are present, \texttt{raise} re-raises the last exception that was active in the current scope. If no exception is active in the current scope, a \texttt{TypeError} exception is raised indicating that this is an error (if running under IDLE, a \texttt{Queue.Empty} exception is raised instead).

Otherwise, \texttt{raise} evaluates the expressions to get three objects, using \texttt{None} as the value of omitted expressions. The first two objects are used to determine the \texttt{type} and \texttt{value} of the exception.

If the first object is an instance, the type of the exception is the class of the instance, the instance itself is the value, and the second object must be \texttt{None}.

If the first object is a class, it becomes the type of the exception. The second object is used to determine the exception value: If it is an instance of the class, the instance becomes the exception value. If the second object is a tuple, it is used as the argument list for the class constructor; if it is \texttt{None}, an empty argument list is used, and any other object is treated as a single argument to the constructor. The instance so created by calling the constructor is used as the exception value.

If a third object is present and not \texttt{None}, it must be a traceback object (see section \texttt{The standard type hierarchy}), and it is substituted instead of the current location as the place where the exception occurred. If the third object is present and not a traceback object or \texttt{None}, a \texttt{TypeError} exception is raised. The three-expression form of
`raise` is useful to re-raise an exception transparently in an except clause, but `raise` with no expressions should be preferred if the exception to be re-raised was the most recently active exception in the current scope.

Additional information on exceptions can be found in section `Exceptions`, and information about handling exceptions is in section `The try statement`. 
6.10. The `break` statement

`break_stmt ::= "break"

`break` may only occur syntactically nested in a `for` or `while` loop, but not nested in a function or class definition within that loop.

It terminates the nearest enclosing loop, skipping the optional `else` clause if the loop has one.

If a `for` loop is terminated by `break`, the loop control target keeps its current value.

When `break` passes control out of a `try` statement with a `finally` clause, that `finally` clause is executed before really leaving the loop.
6.11. The `continue` statement

```
continue_stmt ::= "continue"
```

`continue` may only occur syntactically nested in a `for` or `while` loop, but not nested in a function or class definition or `finally` clause within that loop. It continues with the next cycle of the nearest enclosing loop.

When `continue` passes control out of a `try` statement with a `finally` clause, that `finally` clause is executed before really starting the next loop cycle.
6.12. The `import` statement

Import statements are executed in two steps: (1) find a module, and initialize it if necessary; (2) define a name or names in the local namespace (of the scope where the `import` statement occurs). The statement comes in two forms differing on whether it uses the `from` keyword. The first form (without `from`) repeats these steps for each identifier in the list. The form with `from` performs step (1) once, and then performs step (2) repeatedly.

To understand how step (1) occurs, one must first understand how Python handles hierarchical naming of modules. To help organize modules and provide a hierarchy in naming, Python has a concept of packages. A package can contain other packages and modules while modules cannot contain other modules or packages. From a file system perspective, packages are directories and modules are files.

Once the name of the module is known (unless otherwise specified, the term “module” will refer to both packages and modules), searching for the module or package can begin. The first place checked is `sys.modules`, the cache of all modules that have been imported previously. If the module is found there then it is used in step (2) of import.
If the module is not found in the cache, then `sys.meta_path` is searched (the specification for `sys.meta_path` can be found in PEP 302). The object is a list of `finder` objects which are queried in order as to whether they know how to load the module by calling their `find_module()` method with the name of the module. If the module happens to be contained within a package (as denoted by the existence of a dot in the name), then a second argument to `find_module()` is given as the value of the `__path__` attribute from the parent package (everything up to the last dot in the name of the module being imported). If a finder can find the module it returns a `loader` (discussed later) or returns `None`.

If none of the finders on `sys.meta_path` are able to find the module then some implicitly defined finders are queried. Implementations of Python vary in what implicit meta path finders are defined. The one they all do define, though, is one that handles `sys.path_hooks`, `sys.path_importer_cache`, and `sys.path`.

The implicit finder searches for the requested module in the “paths” specified in one of two places (“paths” do not have to be file system paths). If the module being imported is supposed to be contained within a package then the second argument passed to `find_module()`, `__path__` on the parent package, is used as the source of paths. If the module is not contained in a package then `sys.path` is used as the source of paths.

Once the source of paths is chosen it is iterated over to find a finder that can handle that path. The dict at `sys.path_importer_cache` caches finders for paths and is checked for a finder. If the path does not have a finder cached then `sys.path_hooks` is searched by calling each object in the list with a single argument of the path, returning a finder or raises `ImportError`. If a finder is returned then it is cached in `sys.path_importer_cache` and then used for that path entry. If no
finder can be found but the path exists then a value of `None` is stored in `sys.path_importer_cache` to signify that an implicit, file-based finder that handles modules stored as individual files should be used for that path. If the path does not exist then a finder which always returns `None` is placed in the cache for the path.

If no finder can find the module then `ImportError` is raised. Otherwise some finder returned a loader whose `load_module()` method is called with the name of the module to load (see PEP 302 for the original definition of loaders). A loader has several responsibilities to perform on a module it loads. First, if the module already exists in `sys.modules` (a possibility if the loader is called outside of the import machinery) then it is to use that module for initialization and not a new module. But if the module does not exist in `sys.modules` then it is to be added to that dict before initialization begins. If an error occurs during loading of the module and it was added to `sys.modules` it is to be removed from the dict. If an error occurs but the module was already in `sys.modules` it is left in the dict.

The loader must set several attributes on the module. `__name__` is to be set to the name of the module. `__file__` is to be the “path” to the file unless the module is built-in (and thus listed in `sys.builtin_module_names`) in which case the attribute is not set. If what is being imported is a package then `__path__` is to be set to a list of paths to be searched when looking for modules and packages contained within the package being imported. `__package__` is optional but should be set to the name of package that contains the module or package (the empty string is used for module not contained in a package). `__loader__` is also optional but should be set to the loader object that is loading the module.

If an error occurs during loading then the loader raises `ImportError` if some other exception is not already being propagated. Otherwise
the loader returns the module that was loaded and initialized.

When step (1) finishes without raising an exception, step (2) can begin.

The first form of `import` statement binds the module name in the local namespace to the module object, and then goes on to import the next identifier, if any. If the module name is followed by `as`, the name following `as` is used as the local name for the module.

The `from` form does not bind the module name: it goes through the list of identifiers, looks each one of them up in the module found in step (1), and binds the name in the local namespace to the object thus found. As with the first form of `import`, an alternate local name can be supplied by specifying “`as` localname”. If a name is not found, `ImportError` is raised. If the list of identifiers is replaced by a star (`'*'`), all public names defined in the module are bound in the local namespace of the `import` statement..

The public names defined by a module are determined by checking the module’s namespace for a variable named `__all__`; if defined, it must be a sequence of strings which are names defined or imported by that module. The names given in `__all__` are all considered public and are required to exist. If `__all__` is not defined, the set of public names includes all names found in the module’s namespace which do not begin with an underscore character (`'_'`). `__all__` should contain the entire public API. It is intended to avoid accidentally exporting items that are not part of the API (such as library modules which were imported and used within the module).

The `from` form with `*` may only occur in a module scope. If the wild card form of import — `import *` — is used in a function and the function contains or is a nested block with free variables, the
compiler will raise a SyntaxError.

When specifying what module to import you do not have to specify the absolute name of the module. When a module or package is contained within another package it is possible to make a relative import within the same top package without having to mention the package name. By using leading dots in the specified module or package after from you can specify how high to traverse up the current package hierarchy without specifying exact names. One leading dot means the current package where the module making the import exists. Two dots means up one package level. Three dots is up two levels, etc. So if you execute from . import mod from a module in the pkg package then you will end up importing pkg.mod. If you execute from ..subpkg2 import mod from within pkg.subpkg1 you will import pkg.subpkg2.mod. The specification for relative imports is contained within PEP 328.

importlib.import_module() is provided to support applications that determine which modules need to be loaded dynamically.

6.12.1. Future statements

A future statement is a directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python. The future statement is intended to ease migration to future versions of Python that introduce incompatible changes to the language. It allows use of the new features on a per-module basis before the release in which the feature becomes standard.

future_statement ::= "from" "__future__" "import" feature ["as" feature ["as" name]]* | "from" "__future__" "import" "(" feature ["as" name]]* [","] ")"

feature ::= identifier
A future statement must appear near the top of the module. The only lines that can appear before a future statement are:

- the module docstring (if any),
- comments,
- blank lines, and
- other future statements.

The features recognized by Python 2.6 are `unicode_literals`, `print_function`, `absolute_import`, `division`, `generators`, `nested_scopes` and `with_statement`. `generators`, `with_statement`, `nested_scopes` are redundant in Python version 2.6 and above because they are always enabled.

A future statement is recognized and treated specially at compile time: Changes to the semantics of core constructs are often implemented by generating different code. It may even be the case that a new feature introduces new incompatible syntax (such as a new reserved word), in which case the compiler may need to parse the module differently. Such decisions cannot be pushed off until runtime.

For any given release, the compiler knows which feature names have been defined, and raises a compile-time error if a future statement contains a feature not known to it.

The direct runtime semantics are the same as for any import statement: there is a standard module `__future__`, described later, and it will be imported in the usual way at the time the future statement is executed.

The interesting runtime semantics depend on the specific feature enabled by the future statement.
Note that there is nothing special about the statement:

```python
import __future__ [as name]
```

That is not a future statement; it's an ordinary import statement with no special semantics or syntax restrictions.

Code compiled by an `exec` statement or calls to the built-in functions `compile()` and `execfile()` that occur in a module `M` containing a future statement will, by default, use the new syntax or semantics associated with the future statement. This can, starting with Python 2.2 be controlled by optional arguments to `compile()` — see the documentation of that function for details.

A future statement typed at an interactive interpreter prompt will take effect for the rest of the interpreter session. If an interpreter is started with the `-i` option, is passed a script name to execute, and the script includes a future statement, it will be in effect in the interactive session started after the script is executed.

See also:

**PEP 236 - Back to the __future__**

The original proposal for the __future__ mechanism.
6.13. The `global` statement

```
global_stmt ::= "global" identifier ("," identifier)*
```

The `global` statement is a declaration which holds for the entire current code block. It means that the listed identifiers are to be interpreted as globals. It would be impossible to assign to a global variable without `global`, although free variables may refer to globals without being declared global.

Names listed in a `global` statement must not be used in the same code block textually preceding that `global` statement.

Names listed in a `global` statement must not be defined as formal parameters or in a `for` loop control target, `class` definition, function definition, or `import` statement.

**CPython implementation detail:** The current implementation does not enforce the latter two restrictions, but programs should not abuse this freedom, as future implementations may enforce them or silently change the meaning of the program.

**Programmer’s note:** the `global` is a directive to the parser. It applies only to code parsed at the same time as the `global` statement. In particular, a `global` statement contained in an `exec` statement does not affect the code block containing the `exec` statement, and code contained in an `exec` statement is unaffected by `global` statements in the code containing the `exec` statement. The same applies to the `eval()`, `execfile()` and `compile()` functions.
6.14. The **exec** statement

This statement supports dynamic execution of Python code. The first expression should evaluate to either a Unicode string, a *Latin-1* encoded string, an open file object, a code object, or a tuple. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs). [1] If it is an open file, the file is parsed until EOF and executed. If it is a code object, it is simply executed. For the interpretation of a tuple, see below. In all cases, the code that’s executed is expected to be valid as file input (see section File input). Be aware that the `return` and `yield` statements may not be used outside of function definitions even within the context of code passed to the `exec` statement.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only the first expression after `in` is specified, it should be a dictionary, which will be used for both the global and the local variables. If two expressions are given, they are used for the global and local variables, respectively. If provided, `locals` can be any mapping object. Remember that at module level, globals and locals are the same dictionary. If two separate objects are given as `globals` and `locals`, the code will be executed as if it were embedded in a class definition.

The first expression may also be a tuple of length 2 or 3. In this case, the optional parts must be omitted. The form `exec(expr, globals)` is equivalent to `exec expr in globals`, while the form `exec(expr, globals, locals)` is equivalent to `exec expr in globals, locals`. The tuple form of `exec` provides compatibility with Python 3, where `exec` is a function rather than a statement.
Changed in version 2.4: Formerly, `locals` was required to be a dictionary.

As a side effect, an implementation may insert additional keys into the dictionaries given besides those corresponding to variable names set by the executed code. For example, the current implementation may add a reference to the dictionary of the built-in module `__builtin__` under the key `__builtins__` (!).

**Programmer’s hints:** dynamic evaluation of expressions is supported by the built-in function `eval()`. The built-in functions `globals()` and `locals()` return the current global and local dictionary, respectively, which may be useful to pass around for use by `exec`.

**Footnotes**

[1] Note that the parser only accepts the Unix-style end of line convention. If you are reading the code from a file, make sure to use universal newlines mode to convert Windows or Mac-style newlines.
7. Compound statements

Compound statements contain (groups of) other statements; they affect or control the execution of those other statements in some way. In general, compound statements span multiple lines, although in simple incarnations a whole compound statement may be contained in one line.

The \texttt{if}, \texttt{while} and \texttt{for} statements implement traditional control flow constructs. \texttt{try} specifies exception handlers and/or cleanup code for a group of statements. Function and class definitions are also syntactically compound statements.

Compound statements consist of one or more ‘clauses.’ A clause consists of a header and a ‘suite.’ The clause headers of a particular compound statement are all at the same indentation level. Each clause header begins with a uniquely identifying keyword and ends with a colon. A suite is a group of statements controlled by a clause. A suite can be one or more semicolon-separated simple statements on the same line as the header, following the header’s colon, or it can be one or more indented statements on subsequent lines. Only the latter form of suite can contain nested compound statements; the following is illegal, mostly because it wouldn’t be clear to which \texttt{if} clause a following \texttt{else} clause would belong:

\begin{verbatim}
if test1: if test2: print x
\end{verbatim}

Also note that the semicolon binds tighter than the colon in this context, so that in the following example, either all or none of the \texttt{print} statements are executed:

\begin{verbatim}
if x < y < z: print x; print y; print z
\end{verbatim}

Summarizing:
compound_stmt ::= if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef | classdef | decorated

suite ::= stmt_list NEWLINE | NEWLINE INDENT statement
statement ::= stmt_list NEWLINE | compound_stmt
stmt_list ::= simple_stmt (";" simple_stmt)* [";;]

Note that statements always end in a `NEWLINE` possibly followed by a `DEDENT`. Also note that optional continuation clauses always begin with a keyword that cannot start a statement, thus there are no ambiguities (the ‘dangling else’ problem is solved in Python by requiring nested `if` statements to be indented).

The formatting of the grammar rules in the following sections places each clause on a separate line for clarity.
7.1. The if statement

The if statement is used for conditional execution:

\[
\text{if_stmt ::= } \text{"if" expression ":" suite} \\
\ \\
\ \\
\ \\
\ \\
\ \\
\ \\
\ \\
\ \\
\ \\
\ \\
\ \\
\ \\
\]  

( "elif" expression ":" suite )*  
[ "else" ":" suite ]

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true (see section Boolean operations for the definition of true and false); then that suite is executed (and no other part of the if statement is executed or evaluated). If all expressions are false, the suite of the else clause, if present, is executed.
7.2. The **while** statement

The **while** statement is used for repeated execution as long as an expression is true:

```
while_stmt ::= "while" expression ":" suite
             ["else" ":" suite]
```

This repeatedly tests the expression and, if it is true, executes the first suite; if the expression is false (which may be the first time it is tested) the suite of the **else** clause, if present, is executed and the loop terminates.

A **break** statement executed in the first suite terminates the loop without executing the **else** clause's suite. A **continue** statement executed in the first suite skips the rest of the suite and goes back to testing the expression.
7.3. The for statement

The `for` statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:

```
for_stmt ::= "for" target_list "in" expression_list ":" suite
            ["else ":" suite]
```

The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the `expression_list`. The suite is then executed once for each item provided by the iterator, in the order of ascending indices. Each item in turn is assigned to the target list using the standard rules for assignments, and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty), the suite in the `else` clause, if present, is executed, and the loop terminates.

A `break` statement executed in the first suite terminates the loop without executing the `else` clause’s suite. A `continue` statement executed in the first suite skips the rest of the suite and continues with the next item, or with the `else` clause if there was no next item.

The suite may assign to the variable(s) in the target list; this does not affect the next item assigned to it.

The target list is not deleted when the loop is finished, but if the sequence is empty, it will not have been assigned to at all by the loop. Hint: the built-in function `range()` returns a sequence of integers suitable to emulate the effect of Pascal's `for i := a to b do`; e.g., `range(3)` returns the list `[0, 1, 2].`

Note: There is a subtlety when the sequence is being modified by
the loop (this can only occur for mutable sequences, i.e. lists). An internal counter is used to keep track of which item is used next, and this is incremented on each iteration. When this counter has reached the length of the sequence the loop terminates. This means that if the suite deletes the current (or a previous) item from the sequence, the next item will be skipped (since it gets the index of the current item which has already been treated). Likewise, if the suite inserts an item in the sequence before the current item, the current item will be treated again the next time through the loop. This can lead to nasty bugs that can be avoided by making a temporary copy using a slice of the whole sequence, e.g.,

```python
for x in a[:]
    if x < 0: a.remove(x)
```
7.4. The `try` statement

The `try` statement specifies exception handlers and/or cleanup code for a group of statements:

```plaintext
try_stmt ::= try1_stmt | try2_stmt
try1_stmt ::= "try" ":" suite
             ("except" [expression [("as" | ",") identifier]]
              ["else" ":" suite]
              ["finally" ":" suite]
try2_stmt ::= "try" ":" suite
             "finally" ":" suite
```

*Changed in version 2.5:* In previous versions of Python, `try...except...finally` did not work. `try...except` had to be nested in `try...finally`.

The `except` clause(s) specify one or more exception handlers. When no exception occurs in the `try` clause, no exception handler is executed. When an exception occurs in the `try` suite, a search for an exception handler is started. This search inspects the except clauses in turn until one is found that matches the exception. An expression-less except clause, if present, must be last; it matches any exception. For an except clause with an expression, that expression is evaluated, and the clause matches the exception if the resulting object is “compatible” with the exception. An object is compatible with an exception if it is the class or a base class of the exception object, or a tuple containing an item compatible with the exception.

If no except clause matches the exception, the search for an exception handler continues in the surrounding code and on the invocation stack. [1]
If the evaluation of an expression in the header of an except clause raises an exception, the original search for a handler is canceled and a search starts for the new exception in the surrounding code and on the call stack (it is treated as if the entire `try` statement raised the exception).

When a matching except clause is found, the exception is assigned to the target specified in that except clause, if present, and the except clause's suite is executed. All except clauses must have an executable block. When the end of this block is reached, execution continues normally after the entire try statement. (This means that if two nested handlers exist for the same exception, and the exception occurs in the try clause of the inner handler, the outer handler will not handle the exception.)

Before an except clause's suite is executed, details about the exception are assigned to three variables in the `sys` module:
- `sys.exc_type` receives the object identifying the exception;
- `sys.exc_value` receives the exception’s parameter;
- `sys.exc_traceback` receives a traceback object (see section *The standard type hierarchy*) identifying the point in the program where the exception occurred. These details are also available through the `sys.exc_info()` function, which returns a tuple `(exc_type, exc_value, exc_traceback)`. Use of the corresponding variables is deprecated in favor of this function, since their use is unsafe in a threaded program. As of Python 1.5, the variables are restored to their previous values (before the call) when returning from a function that handled an exception.

The optional `else` clause is executed if and when control flows off the end of the `try` clause. [2] Exceptions in the `else` clause are not handled by the preceding `except` clauses.

If `finally` is present, it specifies a ‘cleanup’ handler. The `try` clause
is executed, including any `except` and `else` clauses. If an exception occurs in any of the clauses and is not handled, the exception is temporarily saved. The `finally` clause is executed. If there is a saved exception, it is re-raised at the end of the `finally` clause. If the `finally` clause raises another exception or executes a `return` or `break` statement, the saved exception is discarded:

```python
>>> def f():
...     try:
...         1/0
...     finally:
...         return 42
... >>> f()
42
```

The exception information is not available to the program during execution of the `finally` clause.

When a `return`, `break` or `continue` statement is executed in the `try` suite of a `try...finally` statement, the `finally` clause is also executed 'on the way out.' A `continue` statement is illegal in the `finally` clause. (The reason is a problem with the current implementation — this restriction may be lifted in the future).

The return value of a function is determined by the last `return` statement executed. Since the `finally` clause always executes, a `return` statement executed in the `finally` clause will always be the last one executed:

```python
>>> def foo():
...     try:
...         return 'try'
...     finally:
...         return 'finally'
... >>> foo()
'finally'
```
Additional information on exceptions can be found in section Exceptions, and information on using the raise statement to generate exceptions may be found in section The raise statement.
7.5. The `with` statement

*New in version 2.5.*

The `with` statement is used to wrap the execution of a block with methods defined by a context manager (see section [With Statement Context Managers](#)). This allows common `try...except...finally` usage patterns to be encapsulated for convenient reuse.

```
with_stmt ::= "with" with_item ("," with_item)* ":" suite
with_item ::= expression ["as" target]
```

The execution of the `with` statement with one “item” proceeds as follows:

1. The context expression (the expression given in the `with_item`) is evaluated to obtain a context manager.
2. The context manager’s `__exit__()` is loaded for later use.
3. The context manager’s `__enter__()` method is invoked.
4. If a target was included in the `with` statement, the return value from `__enter__()` is assigned to it.

**Note:** The `with` statement guarantees that if the `__enter__()` method returns without an error, then `__exit__()` will always be called. Thus, if an error occurs during the assignment to the target list, it will be treated the same as an error occurring within the suite would be. See step 6 below.

5. The suite is executed.
6. The context manager’s `__exit__()` method is invoked. If an
exception caused the suite to be exited, its type, value, and traceback are passed as arguments to `__exit__()`. Otherwise, three `None` arguments are supplied.

If the suite was exited due to an exception, and the return value from the `__exit__()` method was false, the exception is reraised. If the return value was true, the exception is suppressed, and execution continues with the statement following the `with` statement.

If the suite was exited for any reason other than an exception, the return value from `__exit__()` is ignored, and execution proceeds at the normal location for the kind of exit that was taken.

With more than one item, the context managers are processed as if multiple `with` statements were nested:

```python
with A() as a, B() as b:
    suite
```

is equivalent to

```python
with A() as a:
    with B() as b:
        suite
```

**Note:** In Python 2.5, the `with` statement is only allowed when the `with_statement` feature has been enabled. It is always enabled in Python 2.6.

*Changed in version 2.7:* Support for multiple context expressions.

**See also:**
PEP 343 - The “with” statement

The specification, background, and examples for the Python with statement.
A function definition defines a user-defined function object (see section The standard type hierarchy):

A function definition is an executable statement. Its execution binds the function name in the current local namespace to a function object (a wrapper around the executable code for the function). This function object contains a reference to the current global namespace as the global namespace to be used when the function is called.

The function definition does not execute the function body; this gets executed only when the function is called. [3]

A function definition may be wrapped by one or more decorator expressions. Decorator expressions are evaluated when the function is defined, in the scope that contains the function definition. The result must be a callable, which is invoked with the function object as the only argument. The returned value is bound to the function name instead of the function object. Multiple decorators are applied in nested fashion. For example, the following code:
is equivalent to:

```python
def func(): pass
func = f1(arg)(f2(func))
```

When one or more top-level parameters have the form `parameter = expression`, the function is said to have “default parameter values.” For a parameter with a default value, the corresponding argument may be omitted from a call, in which case the parameter’s default value is substituted. If a parameter has a default value, all following parameters must also have a default value — this is a syntactic restriction that is not expressed by the grammar.

**Default parameter values are evaluated when the function definition is executed.** This means that the expression is evaluated once, when the function is defined, and that the same “pre-computed” value is used for each call. This is especially important to understand when a default parameter is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default value is in effect modified. This is generally not what was intended. A way around this is to use `None` as the default, and explicitly test for it in the body of the function, e.g.:

```python
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Function call semantics are described in more detail in section Calls. A function call always assigns values to all parameters mentioned in the parameter list, either from position arguments, from keyword
arguments, or from default values. If the form “*identifier” is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form “**identifier” is present, it is initialized to a new dictionary receiving any excess keyword arguments, defaulting to a new empty dictionary.

It is also possible to create anonymous functions (functions not bound to a name), for immediate use in expressions. This uses lambda expressions, described in section Lambdas. Note that the lambda expression is merely a shorthand for a simplified function definition; a function defined in a “def” statement can be passed around or assigned to another name just like a function defined by a lambda expression. The “def” form is actually more powerful since it allows the execution of multiple statements.

**Programmer’s note:** Functions are first-class objects. A “def” form executed inside a function definition defines a local function that can be returned or passed around. Free variables used in the nested function can access the local variables of the function containing the def. See section Naming and binding for details.
7.7. Class definitions

A class definition defines a class object (see section The standard type hierarchy):

```
classdef ::= "class" classname [inheritance] ":" suite
inheritance ::= "(" [expression_list] ")"
classname ::= identifier
```

A class definition is an executable statement. It first evaluates the inheritance list, if present. Each item in the inheritance list should evaluate to a class object or class type which allows subclassing. The class's suite is then executed in a new execution frame (see section Naming and binding), using a newly created local namespace and the original global namespace. (Usually, the suite contains only function definitions.) When the class's suite finishes execution, its execution frame is discarded but its local namespace is saved. [4] A class object is then created using the inheritance list for the base classes and the saved local namespace for the attribute dictionary. The class name is bound to this class object in the original local namespace.

**Programmer's note:** Variables defined in the class definition are class variables; they are shared by all instances. To create instance variables, they can be set in a method with `self.name = value`. Both class and instance variables are accessible through the notation “self.name”, and an instance variable hides a class variable with the same name when accessed in this way. Class variables can be used as defaults for instance variables, but using mutable values there can lead to unexpected results. For new-style classes, descriptors can be used to create instance variables with different implementation details.

Class definitions, like function definitions, may be wrapped by one or
more decorator expressions. The evaluation rules for the decorator expressions are the same as for functions. The result must be a class object, which is then bound to the class name.

**Footnotes**

[1] The exception is propagated to the invocation stack unless there is a `finally` clause which happens to raise another exception. That new exception causes the old one to be lost.

Currently, control “flows off the end” except in the case of an exception or the execution of a `return`, `continue`, or `break` statement.

[2] A string literal appearing as the first statement in the function body is transformed into the function's `__doc__` attribute and therefore the function's docstring.

A string literal appearing as the first statement in the class body is transformed into the namespace's `__doc__` item and therefore the class's docstring.
8. Top-level components

The Python interpreter can get its input from a number of sources: from a script passed to it as standard input or as program argument, typed in interactively, from a module source file, etc. This chapter gives the syntax used in these cases.
8.1. Complete Python programs

While a language specification need not prescribe how the language interpreter is invoked, it is useful to have a notion of a complete Python program. A complete Python program is executed in a minimally initialized environment: all built-in and standard modules are available, but none have been initialized, except for `sys` (various system services), `__builtin__` (built-in functions, exceptions and `None`) and `__main__`. The latter is used to provide the local and global namespace for execution of the complete program.

The syntax for a complete Python program is that for file input, described in the next section.

The interpreter may also be invoked in interactive mode; in this case, it does not read and execute a complete program but reads and executes one statement (possibly compound) at a time. The initial environment is identical to that of a complete program; each statement is executed in the namespace of `__main__`.

Under Unix, a complete program can be passed to the interpreter in three forms: with the `-c string` command line option, as a file passed as the first command line argument, or as standard input. If the file or standard input is a tty device, the interpreter enters interactive mode; otherwise, it executes the file as a complete program.
8.2. File input

All input read from non-interactive files has the same form:

```
file_input ::= (NEWLINE | statement)*
```

This syntax is used in the following situations:

- when parsing a complete Python program (from a file or from a string);
- when parsing a module;
- when parsing a string passed to the `exec` statement;
8.3. Interactive input

Input in interactive mode is parsed using the following grammar:

```
interactive_input ::= [stmt_list] NEWLINE | compound_stmt NEWLINE
```

Note that a (top-level) compound statement must be followed by a blank line in interactive mode; this is needed to help the parser detect the end of the input.
8.4. Expression input

There are two forms of expression input. Both ignore leading whitespace. The string argument to `eval()` must have the following form:

```
expr_input ::= expression_list NEWLINE*
```

The input line read by `input()` must have the following form:

```
input_input ::= expression_list NEWLINE
```

Note: to read ‘raw’ input line without interpretation, you can use the built-in function `raw_input()` or the `readline()` method of file objects.
9. Full Grammar specification

This is the full Python grammar, as it is read by the parser generator and used to parse Python source files:

```
# Grammar for Python
# Note: Changing the grammar specified in this file will most likely
# require corresponding changes in the parser module
# (modules/parsermodule.c). If you can't make the changes
# that module yourself, please co-ordinate the required
# with someone who can; ask around on python-dev for help.
# Drake <fdrake@acm.org> will probably be listening there.

# NOTE WELL: You should also follow all the steps listed in PEP
# "How to Change Python's Grammar"

# Start symbols for the grammar:
# single_input is a single interactive statement;
# file_input is a module or sequence of commands read from
# eval_input is the input for the eval() and input() functions
# NB: compound_stmt in single_input is followed by extra NEWLINE
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE*

decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
decorators: decorator+
decorated: decorators (classdef | funcdef)
funcdef: 'def' NAME parameters ':' suite
parameters: '(' [varargslist] ')' 
varargslist: ((fpdef [=] test) ',')*
    ('**' NAME [ ',' '**' NAME] | ***' NAME) |
    fpdef [=] test (',' fpdef [=] test])* [',']
fpdef: NAME | '(' fplist ')
fplist: fpdef (',' fpdef)* [',']

stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | print_stmt | del_stmt | pass_stmt | if_stmt |
    import_stmt | global_stmt | exec_stmt | assert_stmt |
    while_stmt | for_stmt | try_stmt | pass_stmt |
    def_stmt | lambda_stmt | return_stmt | continue_stmt |
    break_stmt | raise_stmt | assert_stmt | local_stmt)
expr_stmt: testlist (augassign (yield_expr|testlist)) |
    ('=' (yield_expr|testlist))
augassign: ('+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' | '<<='| '>>='| '|='| '&='| '<<='| '>>=' | '<=' | '>=' | '==' | '!=' | 'is' | 'is not'
```
For normal assignments, additional restrictions enforced by the interpreter.

```
print_stmt: 'print' ( [ test (',', test)* ['}', ']'] | '>>' test [ (',', test)+ ['}', ']'] )
```

del_stmt: 'del' exprlist

```
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt
break_stmt: 'break'
continue_stmt: 'continue'
return_stmt: 'return' [testlist]
```

```
yield_stmt: yield_expr
raise_stmt: 'raise' [test [',', test [',', test]]
import_stmt: import_name | import_from
```

```
import_name: 'import' dotted_as_names
import_from: ('from' ('.*' dotted_name | '.')+) 'import' ('**' | '(' import_as_names ')') | import_
```

```
import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',', import_as_name)* ['}', ']']
dotted_as_names: dotted_name (',', dotted_as_name)*
dotted_name: NAME ('.*' NAME)
global_stmt: 'global' NAME (',', NAME)*
exec_stmt: 'exec' expr ['in' test [',', test]]
assert_stmt: 'assert' test [',', test]
```

compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt
ifstmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' 'while_stmt: 'while' test ':' suite ['else' '':' suite]
for_stmt: 'for' exprlist 'in' testlist '::' suite ['else' ':' suite] try_stmt: ('try' ':' suite
((except_clause ::' suite)+
  ['else' '':' suite]
  ['finally' '':' suite] |
  'finally' '':' suite))
with_stmt: 'with' with_item (',', with_item)* '':' suite
with_item: test ['as' expr]
```

``` # NB compile.c makes sure that the default except clause is last
except_clause: 'except' [test [('as' | ',') test]]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT
```

```
# Backward compatibility cruft to support:
# [ x for x in lambda: True, lambda: False if x() ]
# even while also allowing:
# lambda x: 5 if x else 2
# (But not a mix of the two)
testlist_safe: old_test [(',', old_test)+ [',']]
old_test: or_test | old_lambdef
```
old_lambdef: 'lambda' [varargslist] ':' old_test

test: or_test ['if' or_test 'else' test] | lambdef
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
comp_op: '<' | '==' | '!=' | '<=' | 'in' | 'not' | 'in' | 'is' |
expr: xor_expr (| xor_expr)*
xor_expr: and_expr (^ and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr ('<<' | '>>') arith_expr*
arith_expr: term (('|' | '-') term)*
term: factor (('*' | '/' | '%' | '//') factor)*
factor: ('+' | '-' | '~') factor | power
power: atom trailer* [***] factor
atom: ('(' ['yield_expr|testlist_comp] ')' |
'[' ['listmaker'] ']' |
'{' ['dictorsetmaker'] '}' |
'' testlist1 ' ' |
NAME | NUMBER | STRING+)
listmaker: test ( list_for | (', test)* [','] )
testlist_comp: test ( comp_for | (', test)* [','] )
lambdef: 'lambda' [varargslist] ':' test
trailer: (['arglist'] ')' | ['subscriptlist '] | '.' NAME
subscriptlist: subscript (', subscript)* [',']
subscript: '.' '.' '.' | test | [test] ':=' [test] [sliceop]
sliceop: ':' [test]
exprlist: expr (',' expr)* ['',']
testlist: test (',' test)* ['',']
dictorsetmaker: (test ':' test (comp_for | (', test ':' test (test (comp_for | (', test)* ['',']))))

classdef: 'class' NAME [('[' [testlist] '']) '':' suite

arglist: (argument ',')* (argument ['',']
|'**' test (',' argument)* ['',' '***' test |
|'***' test)

# The reason that keywords are test nodes instead of NAME is that # results in an ambiguity. ast.c makes sure it's a NAME.
argument: test [comp_for] | test '=' test

list_iter: list_for | list_if
list_for: 'for' exprlist 'in' testlist_safe [list_iter]
list_if: 'if' old_test [list_iter]

comp_iter: comp_for | comp_if
comp_for: 'for' exprlist 'in' or_test [comp_iter]
comp_if: 'if' old_test [comp_iter]

testlist1: test (',', test)*

# not used in grammar, but may appear in "node" passed from Par
encoding_decl: NAME

yield_expr: 'yield' [testlist]
1. Introduction

The “Python library” contains several different kinds of components.

It contains data types that would normally be considered part of the “core” of a language, such as numbers and lists. For these types, the Python language core defines the form of literals and places some constraints on their semantics, but does not fully define the semantics. (On the other hand, the language core does define syntactic properties like the spelling and priorities of operators.)

The library also contains built-in functions and exceptions — objects that can be used by all Python code without the need of an `import` statement. Some of these are defined by the core language, but many are not essential for the core semantics and are only described here.

The bulk of the library, however, consists of a collection of modules. There are many ways to dissect this collection. Some modules are written in C and built in to the Python interpreter; others are written in Python and imported in source form. Some modules provide interfaces that are highly specific to Python, like printing a stack trace; some provide interfaces that are specific to particular operating systems, such as access to specific hardware; others provide interfaces that are specific to a particular application domain, like the World Wide Web. Some modules are available in all versions and ports of Python; others are only available when the underlying system supports or requires them; yet others are available only when a particular configuration option was chosen at the time when Python was compiled and installed.

This manual is organized “from the inside out:” it first describes the built-in data types, then the built-in functions and exceptions, and finally the modules, grouped in chapters of related modules. The
ordering of the chapters as well as the ordering of the modules within each chapter is roughly from most relevant to least important.

This means that if you start reading this manual from the start, and skip to the next chapter when you get bored, you will get a reasonable overview of the available modules and application areas that are supported by the Python library. Of course, you don’t have to read it like a novel — you can also browse the table of contents (in front of the manual), or look for a specific function, module or term in the index (in the back). And finally, if you enjoy learning about random subjects, you choose a random page number (see module `random`) and read a section or two. Regardless of the order in which you read the sections of this manual, it helps to start with chapter `Built-in Functions`, as the remainder of the manual assumes familiarity with this material.

Let the show begin!
2. Built-in Functions

The Python interpreter has a number of functions built into it that are always available. They are listed here in alphabetical order.

<table>
<thead>
<tr>
<th>Built-in Functions</th>
<th>Built-in Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs()</td>
<td>divmod()</td>
</tr>
<tr>
<td>all()</td>
<td>enumerate()</td>
</tr>
<tr>
<td>any()</td>
<td>eval()</td>
</tr>
<tr>
<td>basestring()</td>
<td>execfile()</td>
</tr>
<tr>
<td>bin()</td>
<td>file()</td>
</tr>
<tr>
<td>bool()</td>
<td>filter()</td>
</tr>
<tr>
<td>bytearray()</td>
<td>float()</td>
</tr>
<tr>
<td>callable()</td>
<td>format()</td>
</tr>
<tr>
<td>chr()</td>
<td>frozenset()</td>
</tr>
<tr>
<td>classmethod()</td>
<td>getattr()</td>
</tr>
<tr>
<td>cmp()</td>
<td>globals()</td>
</tr>
<tr>
<td>complex()</td>
<td>hash()</td>
</tr>
<tr>
<td>delattr()</td>
<td>help()</td>
</tr>
<tr>
<td>dict()</td>
<td>hex()</td>
</tr>
<tr>
<td>dir()</td>
<td>id()</td>
</tr>
<tr>
<td>el()</td>
<td>input()</td>
</tr>
<tr>
<td>open()</td>
<td>int()</td>
</tr>
<tr>
<td>ord()</td>
<td>isinstance()</td>
</tr>
<tr>
<td>pow()</td>
<td>iter()</td>
</tr>
<tr>
<td>property()</td>
<td>list()</td>
</tr>
<tr>
<td>range()</td>
<td>raw_input()</td>
</tr>
<tr>
<td>set()</td>
<td>repr()</td>
</tr>
<tr>
<td>setattr()</td>
<td>reversed()</td>
</tr>
<tr>
<td>slice()</td>
<td>round()</td>
</tr>
<tr>
<td>sorted()</td>
<td><strong>import</strong>()</td>
</tr>
</tbody>
</table>

In addition, there are other four built-in functions that are no longer considered essential: apply(), buffer(), coerce(), and intern(). They are documented in the Non-essential Built-in Functions section.

abs(x)

Return the absolute value of a number. The argument may be a plain or long integer or a floating point number. If the argument is a complex number, its magnitude is returned.

all(iterable)

Return True if all elements of the iterable are true (or if the iterable is empty). Equivalent to:
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True

New in version 2.5.

any(iterable)
Return True if any element of the iterable is true. If the iterable is empty, return False. Equivalent to:

def any(iterable):
    for element in iterable:
        if element:
            return True
    return False

New in version 2.5.

basestring()
This abstract type is the superclass for str and unicode. It cannot be called or instantiated, but it can be used to test whether an object is an instance of str or unicode. isinstance(obj, basestring) is equivalent to isinstance(obj, (str, unicode)).

New in version 2.3.

bin(x)
Convert an integer number to a binary string. The result is a valid Python expression. If x is not a Python int object, it has to define an __index__() method that returns an integer.

New in version 2.6.

class bool([x])
Return a Boolean value, i.e. one of True or False. x is converted
using the standard truth testing procedure. If \( x \) is false or omitted, this returns \texttt{False} ; otherwise it returns \texttt{True}. \texttt{bool} is also a class, which is a subclass of \texttt{int}. Class \texttt{bool} cannot be subclassed further. Its only instances are \texttt{False} and \texttt{True}.

*New in version 2.2.1.*

*Changed in version 2.3:* If no argument is given, this function returns \texttt{False}.

```
class bytearray([source[, encoding[, errors]]])
```

Return a new array of bytes. The \texttt{bytearray} class is a mutable sequence of integers in the range \( 0 \leq x < 256 \). It has most of the usual methods of mutable sequences, described in \texttt{Mutable Sequence Types}, as well as most methods that the \texttt{str} type has, see \texttt{String Methods}.

The optional \texttt{source} parameter can be used to initialize the array in a few different ways:

- If it is \texttt{unicode}, you must also give the \texttt{encoding} (and optionally, \texttt{errors}) parameters; \texttt{bytearray()} then converts the unicode to bytes using \texttt{unicode.encode()}.
- If it is an \texttt{integer}, the array will have that size and will be initialized with null bytes.
- If it is an object conforming to the \texttt{buffer} interface, a read-only buffer of the object will be used to initialize the bytes array.
- If it is an \texttt{iterable}, it must be an iterable of integers in the range \( 0 \leq x < 256 \), which are used as the initial contents of the array.

Without an argument, an array of size 0 is created.

*New in version 2.6.*
callable(object)

Return True if the object argument appears callable, False if not. If this returns true, it is still possible that a call fails, but if it is false, calling object will never succeed. Note that classes are callable (calling a class returns a new instance); class instances are callable if they have a __call__() method.

chr(i)

Return a string of one character whose ASCII code is the integer i. For example, chr(97) returns the string 'a'. This is the inverse of ord(). The argument must be in the range [0..255], inclusive; ValueError will be raised if i is outside that range. See also unichr().

classmethod(function)

Return a class method for function.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```python
class C(object):
    @classmethod
    def f(cls, arg1, arg2, ...):
        ...
```

The @classmethod form is a function decorator – see the description of function definitions in Function definitions for details.

It can be called either on the class (such as c.f()) or on an instance (such as c().f()). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.
Class methods are different than C++ or Java static methods. If you want those, see `staticmethod()` in this section.

For more information on class methods, consult the documentation on the standard type hierarchy in The standard type hierarchy.

*New in version 2.2.*

*Changed in version 2.4:* Function decorator syntax added.

```python
cmp(x, y)
```

Compare the two objects `x` and `y` and return an integer according to the outcome. The return value is negative if `x < y`, zero if `x == y` and strictly positive if `x > y`.

```python
compile(source, filename, mode[, flags[, dont_inherit]])
```

Compile the `source` into a code or AST object. Code objects can be executed by an `exec` statement or evaluated by a call to `eval()`. `source` can either be a Unicode string, a Latin-1 encoded string or an AST object. Refer to the `ast` module documentation for information on how to work with AST objects.

The `filename` argument should give the file from which the code was read; pass some recognizable value if it wasn’t read from a file (`'<string>'` is commonly used).

The `mode` argument specifies what kind of code must be compiled; it can be `'exec'` if `source` consists of a sequence of statements, `'eval'` if it consists of a single expression, or `'single'` if it consists of a single interactive statement (in the latter case, expression statements that evaluate to something other than `None` will be printed).
The optional arguments flags and dont_inherit control which future statements (see PEP 236) affect the compilation of source. If neither is present (or both are zero) the code is compiled with those future statements that are in effect in the code that is calling compile(). If the flags argument is given and dont_inherit is not (or is zero) then the future statements specified by the flags argument are used in addition to those that would be used anyway. If dont_inherit is a non-zero integer then the flags argument is it – the future statements in effect around the call to compile are ignored.

Future statements are specified by bits which can be bitwise ORed together to specify multiple statements. The bitfield required to specify a given feature can be found as the compiler_flag attribute on the _Feature instance in the __future__ module.

This function raises SyntaxError if the compiled source is invalid, and TypeError if the source contains null bytes.

If you want to parse Python code into its AST representation, see ast.parse().

**Note:** When compiling a string with multi-line code in 'single' or 'eval' mode, input must be terminated by at least one newline character. This is to facilitate detection of incomplete and complete statements in the code module.

*Changed in version 2.3:* The flags and dont_inherit arguments were added.

*Changed in version 2.6:* Support for compiling AST objects.

*Changed in version 2.7:* Allowed use of Windows and Mac
newlines. Also input in 'exec' mode does not have to end in a newline anymore.

```python
class complex([real[, imag]])
```

Return a complex number with the value `real + imag*1j` or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If `imag` is omitted, it defaults to zero and the function serves as a numeric conversion function like `int()`, `long()` and `float()`. If both arguments are omitted, returns `0j`.

**Note:** When converting from a string, the string must not contain whitespace around the central `+` or `-` operator. For example, `complex('1+2j')` is fine, but `complex('1 + 2j')` raises `ValueError`.

The complex type is described in **Numeric Types — int, float, long, complex**.

```python
defattr(object, name)
```

This is a relative of `setattr()`. The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, `defattr(x, 'foobare')` is equivalent to `del x.foobare`.

```python
class dict(**kwarg)
class dict(mapping, **kwarg)
class dict(iterable, **kwarg)
```

Create a new dictionary. The `dict` object is the dictionary class.
See `dict` and `Mapping Types — dict` for documentation about this class.

For other containers see the built-in `list`, `set`, and `tuple` classes, as well as the `collections` module.

```python
dir([object])
```

Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object.

If the object has a method named `__dir__()`, this method will be called and must return the list of attributes. This allows objects that implement a custom `__getattr__()` or `__getattribute__()` function to customize the way `dir()` reports their attributes.

If the object does not provide `__dir__()`, the function tries its best to gather information from the object’s `__dict__` attribute, if defined, and from its type object. The resulting list is not necessarily complete, and may be inaccurate when the object has a custom `__getattr__()`. The default `dir()` mechanism behaves differently with different types of objects, as it attempts to produce the most relevant, rather than complete, information:

- If the object is a module object, the list contains the names of the module’s attributes.
- If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.
- Otherwise, the list contains the object’s attributes’ names, the names of its class’s attributes, and recursively of the attributes of its class’s base classes.
The resulting list is sorted alphabetically. For example:

```python
>>> import struct
>>> dir()  # show the names in the module namespace
['__builtins__', '__doc__', '__name__', 'struct']
>>> dir(struct)  # show the names in the struct module
['Struct', '__builtins__', '__doc__', '__file__', '__name__', '__package__', '_clearcache', 'calcsize', 'error', 'pack', 'unpack', 'unpack_from']
>>> class Shape(object):
    def __dir__(self):
        return ['area', 'perimeter', 'location']

>>> s = Shape()
>>> dir(s)
['area', 'perimeter', 'location']
```

**Note:** Because `dir()` is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

`divmod(a, b)`

Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using long division. With mixed operand types, the rules for binary arithmetic operators apply. For plain and long integers, the result is the same as `(a // b, a % b)`. For floating point numbers the result is `(q, a % b)`, where `q` is usually `math.floor(a / b)` but may be 1 less than that. In any case `q * b + a % b` is very close to `a`, if `a % b` is non-zero it has the same sign as `b`, and `0 <= abs(a % b) < abs(b)`.

*Changed in version 2.3:* Using `divmod()` with complex numbers is deprecated.
**enumerate**(*sequence*, *start=0*)

Return an enumerate object. *sequence* must be a sequence, an iterator, or some other object which supports iteration. The `next()` method of the iterator returned by `enumerate()` returns a tuple containing a count (from *start* which defaults to 0) and the values obtained from iterating over *sequence*:

```python
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

Equivalent to:

```python
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
    n += 1
```

*New in version 2.3.*

*Changed in version 2.6:* The *start* parameter was added.

**eval**(*expression[, *globals*[ [, *locals*]]]*)

The arguments are a Unicode or *Latin-1* encoded string and optional globals and locals. If provided, *globals* must be a dictionary. If provided, *locals* can be any mapping object.

*Changed in version 2.4:* formerly *locals* was required to be a dictionary.

The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* dictionaries as global and local namespace. If the *globals* dictionary is present and lacks `__builtins__`, the current globals are copied into *globals* before *expression* is
parsed. This means that `expression` normally has full access to the standard `__builtin__` module and restricted environments are propagated. If the `locals` dictionary is omitted it defaults to the `globals` dictionary. If both dictionaries are omitted, the expression is executed in the environment where `eval()` is called. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```python
>>> x = 1
>>> print eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (such as those created by `compile()`). In this case pass a code object instead of a string. If the code object has been compiled with `'exec' as the `mode` argument, `eval()`'s return value will be `None`.

Hints: dynamic execution of statements is supported by the `exec` statement. Execution of statements from a file is supported by the `execfile()` function. The `globals()` and `locals()` functions returns the current global and local dictionary, respectively, which may be useful to pass around for use by `eval()` or `execfile()`.

See `ast.literal_eval()` for a function that can safely evaluate strings with expressions containing only literals.

`execfile(filename[, globals[, locals]])`

This function is similar to the `exec` statement, but parses a file instead of a string. It is different from the `import` statement in that it does not use the module administration — it reads the file unconditionally and does not create a new module. [1]

The arguments are a file name and two optional dictionaries. The
file is parsed and evaluated as a sequence of Python statements (similarly to a module) using the globals and locals dictionaries as global and local namespace. If provided, locals can be any mapping object. Remember that at module level, globals and locals are the same dictionary. If two separate objects are passed as globals and locals, the code will be executed as if it were embedded in a class definition.

*Changed in version 2.4:* formerly locals was required to be a dictionary.

If the locals dictionary is omitted it defaults to the globals dictionary. If both dictionaries are omitted, the expression is executed in the environment where execfile() is called. The return value is None.

**Note:** The default locals act as described for function locals() below: modifications to the default locals dictionary should not be attempted. Pass an explicit locals dictionary if you need to see effects of the code on locals after function execfile() returns. execfile() cannot be used reliably to modify a function's locals.

```python
file(name[, mode[, buffering]])
```
Constructor function for the file type, described further in section File Objects. The constructor's arguments are the same as those of the open() built-in function described below.

When opening a file, it's preferable to use open() instead of invoking this constructor directly. file is more suited to type testing (for example, writing isinstance(f, file)).

*New in version 2.2.*
filter(function, iterable)

Construct a list from those elements of iterable for which function returns true. iterable may be either a sequence, a container which supports iteration, or an iterator. If iterable is a string or a tuple, the result also has that type; otherwise it is always a list. If function is None, the identity function is assumed, that is, all elements of iterable that are false are removed.

Note that filter(function, iterable) is equivalent to [item for item in iterable if function(item)] if function is not None and [item for item in iterable if item] if function is None.

See itertools.ifilter() and itertools.ifilterfalse() for iterator versions of this function, including a variation that filters for elements where the function returns false.

class float([x])

Return a floating point number constructed from a number or string x.

If the argument is a string, it must contain a possibly signed decimal or floating point number, possibly embedded in whitespace. The argument may also be [+|-]nan or [+|-]inf. Otherwise, the argument may be a plain or long integer or a floating point number, and a floating point number with the same value (within Python's floating point precision) is returned. If no argument is given, returns 0.0.

Note: When passing in a string, values for NaN and Infinity may be returned, depending on the underlying C library. Float accepts the strings nan, inf and -inf for NaN and positive or negative infinity. The case and a leading + are ignored as well as a leading - is ignored for NaN. Float always represents NaN and infinity as nan, inf or -inf.
The float type is described in *Numeric Types — int, float, long, complex*.

**format**(value[, format_spec])

Convert a value to a “formatted” representation, as controlled by `format_spec`. The interpretation of `format_spec` will depend on the type of the value argument, however there is a standard formatting syntax that is used by most built-in types: Format Specification Mini-Language.

```
Note: format(value, format_spec) merely calls value.__format__(format_spec).
```

*New in version 2.6.*

**class frozenset([iterable])**

Return a new frozenset object, optionally with elements taken from iterable. `frozenset` is a built-in class. See `frozenset` and *Set Types — set, frozenset* for documentation about this class.

For other containers see the built-in `set, list, tuple, and dict` classes, as well as the `collections` module.

*New in version 2.4.*

**getattr**(object, name[, default])

Return the value of the named attribute of object. name must be a string. If the string is the name of one of the object’s attributes, the result is the value of that attribute. For example, `getattr(x, 'foobar')` is equivalent to `x.foobar`. If the named attribute does not exist, default is returned if provided, otherwise `AttributeError` is raised.

**globals**()
Return a dictionary representing the current global symbol table. This is always the dictionary of the current module (inside a function or method, this is the module where it is defined, not the module from which it is called).

**hasattr**(object, name)

The arguments are an object and a string. The result is `True` if the string is the name of one of the object’s attributes, `False` if not. (This is implemented by calling `getattr(object, name)` and seeing whether it raises an exception or not.)

**hash**(object)

Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

**help**(object)

Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated.

This function is added to the built-in namespace by the `site` module.

*New in version 2.2.*

**hex**(x)

Convert an integer number (of any size) to a lowercase hexadecimal string prefixed with “0x”, for example:
If x is not a Python int or long object, it has to define an __index__() method that returns an integer.

See also int() for converting a hexadecimal string to an integer using a base of 16.

Note: To obtain a hexadecimal string representation for a float, use the float.hex() method.

Changed in version 2.4: Formerly only returned an unsigned literal.

id(object)
Return the “identity” of an object. This is an integer (or long integer) which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same id() value.

CPython implementation detail: This is the address of the object in memory.

input([prompt])
Equivalent to eval(raw_input(prompt)).

This function does not catch user errors. If the input is not syntactically valid, a SyntaxError will be raised. Other exceptions may be raised if there is an error during evaluation.
If the `readline` module was loaded, then `input()` will use it to provide elaborate line editing and history features.

Consider using the `raw_input()` function for general input from users.

```python
class int(x=0)
class int(x, base=10)
```

Return an integer object constructed from a number or string `x`, or return 0 if no arguments are given. If `x` is a number, it can be a plain integer, a long integer, or a floating point number. If `x` is floating point, the conversion truncates towards zero. If the argument is outside the integer range, the function returns a long object instead.

If `x` is not a number or if `base` is given, then `x` must be a string or Unicode object representing an integer literal in radix `base`. Optionally, the literal can be preceded by + or - (with no space in between) and surrounded by whitespace. A base-n literal consists of the digits 0 to n-1, with a to z (or A to Z) having values 10 to 35. The default `base` is 10. The allowed values are 0 and 2–36. Base-2, -8, and -16 literals can be optionally prefixed with `0b/0B`, `0o/0O`, or `0x/0X`, as with integer literals in code. Base 0 means to interpret the string exactly as an integer literal, so that the actual base is 2, 8, 10, or 16.

The integer type is described in `Numeric Types — int, float, long, complex`.

`isinstance(object, classinfo)`

Return true if the `object` argument is an instance of the `classinfo` argument, or of a (direct, indirect or virtual) subclass thereof. Also return true if `classinfo` is a type object (new-style class) and `object` is an object of that type or of a (direct, indirect or virtual) subclass thereof. If `object` is not a class instance or an object of
the given type, the function always returns false. If `classinfo` is a tuple of class or type objects (or recursively, other such tuples), return true if `object` is an instance of any of the classes or types. If `classinfo` is not a class, type, or tuple of classes, types, and such tuples, a `TypeError` exception is raised.

*Changed in version 2.2:* Support for a tuple of type information was added.

**issubclass(class, classinfo)**
Return true if `class` is a subclass (direct, indirect or virtual) of `classinfo`. A class is considered a subclass of itself. `classinfo` may be a tuple of class objects, in which case every entry in `classinfo` will be checked. In any other case, a `TypeError` exception is raised.

*Changed in version 2.3:* Support for a tuple of type information was added.

**iter(o[, sentinel])**
Return an iterator object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, `o` must be a collection object which supports the iteration protocol (the `__iter__()` method), or it must support the sequence protocol (the `__getitem__()` method with integer arguments starting at 0). If it does not support either of those protocols, `TypeError` is raised. If the second argument, `sentinel`, is given, then `o` must be a callable object. The iterator created in this case will call `o` with no arguments for each call to its `next()` method; if the value returned is equal to `sentinel`, `StopIteration` will be raised, otherwise the value will be returned.

One useful application of the second form of `iter()` is to read lines of a file until a certain line is reached. The following example
reads a file until the `readline()` method returns an empty string:

```python
with open('mydata.txt') as fp:
    for line in iter(fp.readline, ' '):
        process_line(line)
```

*New in version 2.2.*

`len(s)`
Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

`class list([iterable])`
Return a list whose items are the same and in the same order as `iterable`'s items. `iterable` may be either a sequence, a container that supports iteration, or an iterator object. If `iterable` is already a list, a copy is made and returned, similar to `iterable[:]`. For instance, `list('abc')` returns `['a', 'b', 'c']` and `list( (1, 2, 3) )` returns `[1, 2, 3]`. If no argument is given, returns a new empty list, `[]`.

`list` is a mutable sequence type, as documented in `Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange`. For other containers see the built in `dict, set, and tuple` classes, and the `collections` module.

`locals()`
Update and return a dictionary representing the current local symbol table. Free variables are returned by `locals()` when it is called in function blocks, but not in class blocks.

**Note:** The contents of this dictionary should not be modified; changes may not affect the values of local and free variables
used by the interpreter.

```python
class long(x=0)
class long(x, base=10)
```

Return a long integer object constructed from a string or number `x`. If the argument is a string, it must contain a possibly signed number of arbitrary size, possibly embedded in whitespace. The `base` argument is interpreted in the same way as for `int()`, and may only be given when `x` is a string. Otherwise, the argument may be a plain or long integer or a floating point number, and a long integer with the same value is returned. Conversion of floating point numbers to integers truncates (towards zero). If no arguments are given, returns `0L`.

The long type is described in **Numeric Types — int, float, long, complex**.

```python
map(function, iterable, ...)
```

Apply `function` to every item of `iterable` and return a list of the results. If additional `iterable` arguments are passed, `function` must take that many arguments and is applied to the items from all iterables in parallel. If one iterable is shorter than another it is assumed to be extended with `None` items. If `function` is `None`, the identity function is assumed; if there are multiple arguments, `map()` returns a list consisting of tuples containing the corresponding items from all iterables (a kind of transpose operation). The `iterable` arguments may be a sequence or any iterable object; the result is always a list.

```python
max(iterable[, key])
max(arg1, arg2, *args[, key])
```

Return the largest item in an iterable or the largest of two or more arguments.
If one positional argument is provided, *iterable* must be a non-empty iterable (such as a non-empty string, tuple or list). The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

The optional *key* argument specifies a one-argument ordering function like that used for *list.sort()*(). The *key* argument, if supplied, must be in keyword form (for example, max(a, b, c, key=func)).

*Changed in version 2.5:* Added support for the optional *key* argument.

**memoryview**(obj)

Return a “memory view” object created from the given argument. See memoryview type for more information.

**min**(iterable[, key])

**min**(arg1, arg2, *args[, key])

Return the smallest item in an iterable or the smallest of two or more arguments.

If one positional argument is provided, *iterable* must be a non-empty iterable (such as a non-empty string, tuple or list). The smallest item in the iterable is returned. If two or more positional arguments are provided, the smallest of the positional arguments is returned.

The optional *key* argument specifies a one-argument ordering function like that used for *list.sort()*(). The *key* argument, if supplied, must be in keyword form (for example, min(a, b, c, key=func)).

*Changed in version 2.5:* Added support for the optional *key* argument.
argument.

\textbf{next}(\texttt{iterator[, default]})

Retrieve the next item from the \textit{iterator} by calling its \texttt{next()} method. If \texttt{default} is given, it is returned if the iterator is exhausted, otherwise \texttt{StopIteration} is raised.

\textit{New in version 2.6.}

class \texttt{object}

Return a new featureless object. \texttt{object} is a base for all new style classes. It has the methods that are common to all instances of new style classes.

\textit{New in version 2.2.}

\textit{Changed in version 2.3:} This function does not accept any arguments. Formerly, it accepted arguments but ignored them.

\texttt{oct}(\texttt{x})

Convert an integer number (of any size) to an octal string. The result is a valid Python expression.

\textit{Changed in version 2.4:} Formerly only returned an unsigned literal.

\texttt{open}(\texttt{name[, mode[, buffering]]})

Open a file, returning an object of the \texttt{file} type described in section \texttt{File Objects}. If the file cannot be opened, \texttt{IOError} is raised. When opening a file, it's preferable to use \texttt{open()} instead of invoking the \texttt{file} constructor directly.

The first two arguments are the same as for \texttt{stdio}'s \texttt{fopen():} \texttt{name} is the file name to be opened, and \texttt{mode} is a string
indicating how the file is to be opened.

The most commonly-used values of mode are 'r' for reading, 'w' for writing (truncating the file if it already exists), and 'a' for appending (which on some Unix systems means that all writes append to the end of the file regardless of the current seek position). If mode is omitted, it defaults to 'r'. The default is to use text mode, which may convert '\n' characters to a platform-specific representation on writing and back on reading. Thus, when opening a binary file, you should append 'b' to the mode value to open the file in binary mode, which will improve portability. (Appending 'b' is useful even on systems that don't treat binary and text files differently, where it serves as documentation.) See below for more possible values of mode.

The optional buffering argument specifies the file's desired buffer size: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size (in bytes). A negative buffering means to use the system default, which is usually line buffered for tty devices and fully buffered for other files. If omitted, the system default is used. [2]

Modes 'r+', 'w+' and 'a+' open the file for updating (reading and writing); note that 'w+' truncates the file. Append 'b' to the mode to open the file in binary mode, on systems that differentiate between binary and text files; on systems that don't have this distinction, adding the 'b' has no effect.

In addition to the standard fopen() values mode may be 'U' or 'rU'. Python is usually built with universal newlines support; supplying 'U' opens the file as a text file, but lines may be terminated by any of the following: the Unix end-of-line convention '\n', the Macintosh convention '\r', or the Windows
convention 'r\n'. All of these external representations are seen as 'n' by the Python program. If Python is built without universal newlines support a mode with 'u' is the same as normal text mode. Note that file objects so opened also have an attribute called newlines which has a value of None (if no newlines have yet been seen), 'n', 'r', 'rn', or a tuple containing all the newline types seen.

Python enforces that the mode, after stripping 'u', begins with 'r', 'w' or 'a'.

Python provides many file handling modules including fileinput, os, os.path, tempfile, and shutil.

*Changed in version 2.5:* Restriction on first letter of mode string introduced.

**ord(c)**

Given a string of length one, return an integer representing the Unicode code point of the character when the argument is a unicode object, or the value of the byte when the argument is an 8-bit string. For example, ord('a') returns the integer 97, ord(u'\u2020') returns 8224. This is the inverse of chr() for 8-bit strings and of unichr() for unicode objects. If a unicode argument is given and Python was built with UCS2 Unicode, then the character's code point must be in the range [0..65535] inclusive; otherwise the string length is two, and a TypeError will be raised.

**pow(x, y[, z])**

Return x to the power y; if z is present, return x to the power y, modulo z (computed more efficiently than pow(x, y) % z). The two-argument form pow(x, y) is equivalent to using the power
operator: \( x^{**} y \).

The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For int and long int operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, \( 10^{**} 2 \) returns 100, but \( 10^{**} -2 \) returns 0.01. (This last feature was added in Python 2.2. In Python 2.1 and before, if both arguments were of integer types and the second argument was negative, an exception was raised.) If the second argument is negative, the third argument must be omitted. If \( z \) is present, \( x \) and \( y \) must be of integer types, and \( y \) must be non-negative. (This restriction was added in Python 2.2. In Python 2.1 and before, floating 3-argument `pow()` returned platform-dependent results depending on floating-point rounding accidents.)

```python
print(*objects, sep='\ ', end='\n', file=sys.stdout)
```

Print `objects` to the stream `file`, separated by `sep` and followed by `end`. `sep`, `end` and `file`, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by `sep` and followed by `end`. Both `sep` and `end` must be strings; they can also be `None`, which means to use the default values. If no `objects` are given, `print()` will just write `end`.

The `file` argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Output buffering is determined by `file`. Use `file.flush()` to ensure, for instance, immediate appearance on a screen.
Note: This function is not normally available as a built-in since the name print is recognized as the print statement. To disable the statement and use the print() function, use this future statement at the top of your module:

```
from __future__ import print_function
```

New in version 2.6.

class property([fget[, fset[, fdel[, doc]]]])

Return a property attribute for new-style classes (classes that derive from object).

fget is a function for getting an attribute value. fset is a function for setting an attribute value. fdel is a function for deleting an attribute value. And doc creates a docstring for the attribute.

A typical use is to define a managed attribute x:

```python
class C(object):
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

x = property(getx, setx, delx, "I'm the 'x' property.")
```

If c is an instance of C, c.x will invoke the getter, c.x = value will invoke the setter and del c.x the deleter.

If given, doc will be the docstring of the property attribute.
Otherwise, the property will copy `fget`'s docstring (if it exists). This makes it possible to create read-only properties easily using `property()` as a decorator:

```python
class Parrot(object):
    def __init__(self):
        self._voltage = 100000

@property
def voltage(self):
    """Get the current voltage.""
    return self._voltage
```

The `@property` decorator turns the `voltage()` method into a “getter” for a read-only attribute with the same name, and it sets the docstring for `voltage` to “Get the current voltage.”

A property object has `getter`, `setter`, and `deleter` methods usable as decorators that create a copy of the property with the corresponding accessor function set to the decorated function. This is best explained with an example:

```python
class C(object):
    def __init__(self):
        self._x = None

@property
def x(self):
    """I'm the 'x' property.""
    return self._x

@x.setter
def x(self, value):
    self._x = value

@x.deleter
def x(self):
    del self._x
```

This code is exactly equivalent to the first example. Be sure to give the additional functions the same name as the original
property (x in this case.)

The returned property object also has the attributes fget, fset, and fdel corresponding to the constructor arguments.

New in version 2.2.

Changed in version 2.5: Use fget's docstring if no doc given.

Changed in version 2.6: The getter, setter, and deleter attributes were added.

**range(stop)**

**range(start, stop[, step])**

This is a versatile function to create lists containing arithmetic progressions. It is most often used in for loops. The arguments must be plain integers. If the step argument is omitted, it defaults to 1. If the start argument is omitted, it defaults to 0. The full form returns a list of plain integers [start, start + step, start + 2 * step, ...]. If step is positive, the last element is the largest start + i * step less than stop; if step is negative, the last element is the smallest start + i * step greater than stop. step must not be zero (or else ValueError is raised). Example:

```python
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
```

raw_input([prompt])

If the prompt argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, EOFError is raised. Example:

```python
g>>> s = raw_input('--> ') --> Monty Python's Flying Circus g>>> s "Monty Python's Flying Circus"
```

If the readline module was loaded, then raw_input() will use it to provide elaborate line editing and history features.

reduce(function, iterable[, initializer])

Apply function of two arguments cumulatively to the items of iterable, from left to right, so as to reduce the iterable to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates (((((1+2)+3)+4)+5). The left argument, x, is the accumulated value and the right argument, y, is the update value from the iterable. If the optional initializer is present, it is placed before the items of the iterable in the calculation, and serves as a default when the iterable is empty. If initializer is not given and iterable contains only one item, the first item is returned. Roughly equivalent to:

```python
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        try:
            initializer = next(it)
        except StopIteration:
            raise TypeError('reduce() of empty sequence with
    accum_value = initializer
    for x in it:
```
reload(module)
Reload a previously imported module. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (the same as the module argument).

When reload(module) is executed:

- Python modules' code is recompiled and the module-level code reexecuted, defining a new set of objects which are bound to names in the module's dictionary. The init function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects — with a try statement it can test for
the table's presence and skip its initialization if desired:

```python
try:
cache
except NameError:
cache = {}
```

It is generally not very useful to reload built-in or dynamically loaded modules. Reloading `sys`, `__main__`, `builtins` and other key modules is not recommended. In many cases extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from ... import ...`, calling `reload()` for the other module does not redefine the objects imported from it — one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (module.*name*) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

`repr(object)`

Return a string containing a printable representation of an object. This is the same value yielded by conversions (reverse quotes). It is sometimes useful to be able to access this operation as an ordinary function. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()`, otherwise the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a `__repr__()`
method.

**reversed** *(seq)*
Return a reverse iterator. `seq` must be an object which has a `__reversed__()` method or supports the sequence protocol (the `__len__() method and the `__getitem__()` method with integer arguments starting at 0).

*New in version 2.4.*

*Changed in version 2.6:* Added the possibility to write a custom `__reversed__()` method.

**round** *(number[, ndigits])*  
Return the floating point value `number` rounded to `ndigits` digits after the decimal point. If `ndigits` is omitted, it defaults to zero. The result is a floating point number. Values are rounded to the closest multiple of 10 to the power minus `ndigits`; if two multiples are equally close, rounding is done away from 0 (so, for example, `round(0.5)` is `1.0` and `round(-0.5)` is `-1.0`).

**Note:** The behavior of `round()` for floats can be surprising: for example, `round(2.675, 2)` gives `2.67` instead of the expected `2.68`. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. See Floating Point Arithmetic: Issues and Limitations for more information.

**class set([iterable])**  
Return a new `set` object, optionally with elements taken from `iterable`. `set` is a built-in class. See `set` and Set Types — `set`, `frozenset` for documentation about this class.

For other containers see the built-in `frozenset`, `list`, `tuple`, and
dict classes, as well as the collections module.

New in version 2.4.

**setattr(object, name, value)**

This is the counterpart of getattr(). The arguments are an object, a string and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123`.

**class slice(stop)**

**class slice(start, stop[, step])**

Return a slice object representing the set of indices specified by `range(start, stop, step)`. The `start` and `step` arguments default to `None`. Slice objects have read-only data attributes `start`, `stop` and `step` which merely return the argument values (or their default). They have no other explicit functionality; however they are used by Numerical Python and other third party extensions. Slice objects are also generated when extended indexing syntax is used. For example: `a[start:stop:step]` or `a[start:stop, i]`. See `itertools.islice()` for an alternate version that returns an iterator.

**sorted(iterable[, cmp[, key[, reverse]]])**

Return a new sorted list from the items in `iterable`.

The optional arguments `cmp`, `key`, and `reverse` have the same meaning as those for the `list.sort()` method (described in section Mutable Sequence Types).

`cmp` specifies a custom comparison function of two arguments (iterable elements) which should return a negative, zero or positive number depending on whether the first argument is
considered smaller than, equal to, or larger than the second argument: `cmp=lambda x, y: cmp(x.lower(), y.lower())`. The default value is `None`.

`key` specifies a function of one argument that is used to extract a comparison key from each list element: `key=str.lower`. The default value is `None` (compare the elements directly).

`reverse` is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

In general, the `key` and `reverse` conversion processes are much faster than specifying an equivalent `cmp` function. This is because `cmp` is called multiple times for each list element while `key` and `reverse` touch each element only once. Use `functools.cmp_to_key()` to convert an old-style `cmp` function to a `key` function.

The built-in `sorted()` function is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

For sorting examples and a brief sorting tutorial, see [Sorting HOW TO](#).

`New in version 2.4.`

`staticmethod(function)`

Return a static method for `function`.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:
class C(object):
    @staticmethod
    def f(arg1, arg2, ...):
    ...

The @staticmethod form is a function decorator – see the description of function definitions in Function definitions for details.

It can be called either on the class (such as c.f()) or on an instance (such as c().f()). The instance is ignored except for its class.

Static methods in Python are similar to those found in Java or C++. Also see classmethod() for a variant that is useful for creating alternate class constructors.

For more information on static methods, consult the documentation on the standard type hierarchy in The standard type hierarchy.

New in version 2.2.

Changed in version 2.4: Function decorator syntax added.

class str(object='')

Return a string containing a nicely printable representation of an object. For strings, this returns the string itself. The difference with repr(object) is that str(object) does not always attempt to return a string that is acceptable to eval(); its goal is to return a printable string. If no argument is given, returns the empty string, ''.  

For more information on strings see Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange which describes sequence functionality (strings are sequences), and also the
string-specific methods described in the String Methods section. To output formatted strings use template strings or the `%` operator described in the String Formatting Operations section. In addition see the String Services section. See also `unicode()`.

```
sum(iterable[, start])
```

Sums `start` and the items of an `iterable` from left to right and returns the total. `start` defaults to 0. The `iterable`'s items are normally numbers, and the start value is not allowed to be a string.

For some use cases, there are good alternatives to `sum()`. The preferred, fast way to concatenate a sequence of strings is by calling `''.join(sequence)`. To add floating point values with extended precision, see `math.fsum()`. To concatenate a series of iterables, consider using `itertools.chain()`.

*New in version 2.3.*

```
super(type[, object-or-type])
```

Return a proxy object that delegates method calls to a parent or sibling class of `type`. This is useful for accessing inherited methods that have been overridden in a class. The search order is same as that used by `getattr()` except that the `type` itself is skipped.

The `__mro__` attribute of the `type` lists the method resolution search order used by both `getattr()` and `super()`. The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

If the second argument is omitted, the super object returned is unbound. If the second argument is an object, `isinstance(obj,
type) must be true. If the second argument is a type, `issubclass(type2, type)` must be true (this is useful for classmethods).

**Note:** `super()` only works for new-style classes.

There are two typical use cases for `super`. In a class hierarchy with single inheritance, `super` can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of `super` in other programming languages.

The second use case is to support cooperative multiple inheritance in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement “diamond diagrams” where multiple base classes implement the same method. Good design dictates that this method have the same calling signature in every case (because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime).

For both use cases, a typical superclass call looks like this:

```python
class C(B):
    def method(self, arg):
        super(C, self).method(arg)
```

Note that `super()` is implemented as part of the binding process for explicit dotted attribute lookups such as `super().__getitem__(name)`. It does so by implementing its own `__getattr__()` method for searching classes in a predictable order that supports cooperative multiple inheritance.
Accordingly, `super()` is undefined for implicit lookups using statements or operators such as `super()[name]`.

Also note that `super()` is not limited to use inside methods. The two argument form specifies the arguments exactly and makes the appropriate references.

For practical suggestions on how to design cooperative classes using `super()`, see guide to using `super()`.

*New in version 2.2.*

**tuple([`iterable`])**

Return a tuple whose items are the same and in the same order as `iterable`'s items. `iterable` may be a sequence, a container that supports iteration, or an iterator object. If `iterable` is already a tuple, it is returned unchanged. For instance, `tuple('abc')` returns `('a', 'b', 'c')` and `tuple([1, 2, 3])` returns `(1, 2, 3)`. If no argument is given, returns a new empty tuple, `()`.

`tuple` is an immutable sequence type, as documented in Sequence Types — `str`, `unicode`, `list`, `tuple`, `bytearray`, `buffer`, `xrange`. For other containers see the built in `dict`, `list`, and `set` classes, and the `collections` module.

**class type(object)**

**class type(name, bases, dict)**

With one argument, return the type of an `object`. The return value is a type object. The `isinstance()` built-in function is recommended for testing the type of an object.

With three arguments, return a new type object. This is essentially a dynamic form of the `class` statement. The `name` string is the class name and becomes the `__name__` attribute; the
bases tuple itemizes the base classes and becomes the \_bases\_ attribute; and the dict dictionary is the namespace containing definitions for class body and becomes the \_dict\_ attribute. For example, the following two statements create identical type objects:

```python
>>> class X(object):
    ...     a = 1
    ...
>>> X = type('X', (object,), dict(a=1))
```

*New in version 2.2.*

unichr(i)

Return the Unicode string of one character whose Unicode code is the integer \(i\). For example, \texttt{unichr(97)} returns the string \texttt{u'a'}. This is the inverse of \texttt{ord()} for Unicode strings. The valid range for the argument depends on how Python was configured – it may be either UCS2 \([0..0xFFFF]\) or UCS4 \([0..0x10FFFF]\). \texttt{ValueError} is raised otherwise. For ASCII and 8-bit strings see \texttt{chr()}.

*New in version 2.0.*

unicode(object=``)

unicode(object[, encoding[, errors]])

Return the Unicode string version of object using one of the following modes:

If \texttt{encoding} and/or \texttt{errors} are given, \texttt{unicode()} will decode the object which can either be an 8-bit string or a character buffer using the codec for \texttt{encoding}. The \texttt{encoding} parameter is a string giving the name of an encoding; if the encoding is not known, \texttt{LookupError} is raised. Error handling is done according to \texttt{errors}; this specifies the treatment of characters which are invalid in the input encoding. If \texttt{errors} is \texttt{'strict'} (the default), a \texttt{ValueError} is
raised on errors, while a value of 'ignore' causes errors to be silently ignored, and a value of 'replace' causes the official Unicode replacement character, \u+FFFD, to be used to replace input characters which cannot be decoded. See also the codecs module.

If no optional parameters are given, unicode() will mimic the behaviour of str() except that it returns Unicode strings instead of 8-bit strings. More precisely, if object is a Unicode string or subclass it will return that Unicode string without any additional decoding applied.

For objects which provide a __unicode__() method, it will call this method without arguments to create a Unicode string. For all other objects, the 8-bit string version or representation is requested and then converted to a Unicode string using the codec for the default encoding in 'strict' mode.

For more information on Unicode strings see Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange which describes sequence functionality (Unicode strings are sequences), and also the string-specific methods described in the String Methods section. To output formatted strings use template strings or the % operator described in the String Formatting Operations section. In addition see the String Services section. See also str().

New in version 2.0.

Changed in version 2.2: Support for __unicode__() added.

vars([object])
Return the __dict__ attribute for a module, class, instance, or any other object with a __dict__ attribute.
Objects such as modules and instances have an updateable \_dict\_ attribute; however, other objects may have write restrictions on their \_dict\_ attributes (for example, new-style classes use a dictproxy to prevent direct dictionary updates).

Without an argument, \texttt{vars()} acts like \texttt{locals()}. Note, the locals dictionary is only useful for reads since updates to the locals dictionary are ignored.

\texttt{xrange(stop)}
\texttt{xrange(start, stop[, step])}

This function is very similar to \texttt{range()}, but returns an \texttt{xrange} object instead of a list. This is an opaque sequence type which yields the same values as the corresponding list, without actually storing them all simultaneously. The advantage of \texttt{xrange()} over \texttt{range()} is minimal (since \texttt{xrange()} still has to create the values when asked for them) except when a very large range is used on a memory-starved machine or when all of the range’s elements are never used (such as when the loop is usually terminated with \texttt{break}). For more information on xrange objects, see \texttt{XRange Type and Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange}.

\textbf{CPython implementation detail:} \texttt{xrange()} is intended to be simple and fast. Implementations may impose restrictions to achieve this. The C implementation of Python restricts all arguments to native C longs (“short” Python integers), and also requires that the number of elements fit in a native C long. If a larger range is needed, an alternate version can be crafted using the \texttt{itertools} module: \texttt{islice(count(start, step), (stop-start+step-1+2*(step<0))//step)}.  

zip([iterable, ...])

This function returns a list of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables. The returned list is truncated in length to the length of the shortest argument sequence. When there are multiple arguments which are all of the same length, zip() is similar to map() with an initial argument of None. With a single sequence argument, it returns a list of 1-tuples. With no arguments, it returns an empty list.

The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using zip(*[iter(s)]*n).

zip() in conjunction with the * operator can be used to unzip a list:

```python
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> zipped
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zipped)
>>> x == list(x2) and y == list(y2)
True
```

New in version 2.0.

Changed in version 2.4: Formerly, zip() required at least one argument and zip() raised a TypeError instead of returning an empty list.

__import__(name[, globals[, locals[, fromlist[, level]]]]))

Note: This is an advanced function that is not needed in everyday Python programming, unlike
This function is invoked by the `import` statement. It can be replaced (by importing the `__builtin__` module and assigning to `__builtin__.__import__`) in order to change semantics of the `import` statement, but nowadays it is usually simpler to use import hooks (see PEP 302). Direct use of `__import__()` is rare, except in cases where you want to import a module whose name is only known at runtime.

The function imports the module `name`, potentially using the given `globals` and `locals` to determine how to interpret the name in a package context. The `fromlist` gives the names of objects or submodules that should be imported from the module given by `name`. The standard implementation does not use its `locals` argument at all, and uses its `globals` only to determine the package context of the `import` statement.

`level` specifies whether to use absolute or relative imports. The default is `-1` which indicates both absolute and relative imports will be attempted. 0 means only perform absolute imports. Positive values for `level` indicate the number of parent directories to search relative to the directory of the module calling `__import__()`.  

When the `name` variable is of the form `package.module`, normally, the top-level package (the name up till the first dot) is returned, not the module named by `name`. However, when a non-empty `fromlist` argument is given, the module named by `name` is returned.

For example, the statement `import spam` results in bytecode resembling the following code:
The statement `import spam.ham` results in this call:

```python
spam = __import__('spam', globals(), locals(), [], -1)
```

Note how `__import__()` returns the toplevel module here because this is the object that is bound to a name by the `import` statement.

On the other hand, the statement `from spam.ham import eggs, sausage as saus` results in

```python
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], -1)
eggs = _temp.eggs
saus = _temp.sausage
```

Here, the `spam.ham` module is returned from `__import__()`. From this object, the names to import are retrieved and assigned to their respective names.

If you simply want to import a module (potentially within a package) by name, use `importlib.import_module()`.

*Changed in version 2.5*: The level parameter was added.

*Changed in version 2.5*: Keyword support for parameters was added.
There are several built-in functions that are no longer essential to learn, know or use in modern Python programming. They have been kept here to maintain backwards compatibility with programs written for older versions of Python.

Python programmers, trainers, students and book writers should feel free to bypass these functions without concerns about missing something important.

\texttt{apply(function, args[, keywords])}

The \texttt{function} argument must be a callable object (a user-defined or built-in function or method, or a class object) and the \texttt{args} argument must be a sequence. The \texttt{function} is called with \texttt{args} as the argument list; the number of arguments is the length of the tuple. If the optional \texttt{keywords} argument is present, it must be a dictionary whose keys are strings. It specifies keyword arguments to be added to the end of the argument list. Calling \texttt{apply()} is different from just calling \texttt{function(args)}, since in that case there is always exactly one argument. The use of \texttt{apply()} is equivalent to \texttt{function(*args, **keywords)}.

\textit{Deprecated since version 2.3:} Use \texttt{function(*args, **keywords)} instead of \texttt{apply(function, args, keywords)} (see Unpacking Argument Lists).

\texttt{buffer(object[, offset[, size]])}

The \texttt{object} argument must be an object that supports the buffer call interface (such as strings, arrays, and buffers). A new buffer object will be created which references the \texttt{object} argument. The buffer object will be a slice from the beginning of \texttt{object} (or from
the specified offset). The slice will extend to the end of object (or will have a length given by the size argument).

**coerce(x, y)**

Return a tuple consisting of the two numeric arguments converted to a common type, using the same rules as used by arithmetic operations. If coercion is not possible, raise TypeError.

**intern(string)**

Enter string in the table of “interned” strings and return the interned string – which is string itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup – if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys.

*Changed in version 2.3:* Interned strings are not immortal (like they used to be in Python 2.2 and before); you must keep a reference to the return value of intern() around to benefit from it.

**Footnotes**

[1] It is used relatively rarely so does not warrant being made into a statement.

Specifying a buffer size currently has no effect on systems that don’t have setvbuf(). The interface to specify the buffer size is not done using a method that calls setvbuf(), because that may dump core when called after any I/O has been performed, and there’s no reliable way to determine whether this is the case.

In the current implementation, local variable bindings cannot normally be affected this way, but variables retrieved from other scopes (such as modules) can be. This may change.
4. Built-in Constants

A small number of constants live in the built-in namespace. They are:

**False**
The false value of the `bool` type.

*New in version 2.3.*

**True**
The true value of the `bool` type.

*New in version 2.3.*

**None**
The sole value of `types.NoneType`. `None` is frequently used to represent the absence of a value, as when default arguments are not passed to a function.

*Changed in version 2.4: Assignments to `None` are illegal and raise a `SyntaxError`.*

**NotImplemented**
Special value which can be returned by the “rich comparison” special methods (`__eq__`, `__lt__`, and friends), to indicate that the comparison is not implemented with respect to the other type.

**Ellipsis**
Special value used in conjunction with extended slicing syntax.

**__debug__**
This constant is true if Python was not started with an `-o` option. See also the `assert` statement.
Note: The names `None` and `__debug__` cannot be reassigned (assignments to them, even as an attribute name, raise `SyntaxError`), so they can be considered “true” constants.

`Changed in version 2.7:` Assignments to `__debug__` as an attribute became illegal.
4.1. Constants added by the `site` module

The `site` module (which is imported automatically during startup, except if the `-s` command-line option is given) adds several constants to the built-in namespace. They are useful for the interactive interpreter shell and should not be used in programs.

```python
quit([code=None])
exit([code=None])
```

Objects that when printed, print a message like “Use quit() or Ctrl-D (i.e. EOF) to exit”, and when called, raise `SystemExit` with the specified exit code.

```python
copyright
license
credits
```

Objects that when printed, print a message like “Type license() to see the full license text”, and when called, display the corresponding text in a pager-like fashion (one screen at a time).
5. Built-in Types

The following sections describe the standard types that are built into the interpreter.

**Note:** Historically (until release 2.2), Python's built-in types have differed from user-defined types because it was not possible to use the built-in types as the basis for object-oriented inheritance. This limitation no longer exists.

The principal built-in types are numerics, sequences, mappings, files, classes, instances and exceptions.

Some operations are supported by several object types; in particular, practically all objects can be compared, tested for truth value, and converted to a string (with the `repr()` function or the slightly different `str()` function). The latter function is implicitly used when an object is written by the `print()` function.
5.1. Truth Value Testing

Any object can be tested for truth value, for use in an `if` or `while` condition or as operand of the Boolean operations below. The following values are considered false:

- `None`
- `False`
- zero of any numeric type, for example, `0`, `0L`, `0.0`, `0j`.
- any empty sequence, for example, `'`, `()`, `[]`.
- any empty mapping, for example, `{}`.
- instances of user-defined classes, if the class defines a `__nonzero__()` or `__len__()` method, when that method returns the integer zero or `bool` value `False`. [1]

All other values are considered true — so objects of many types are always true.

Operations and built-in functions that have a Boolean result always return `0` or `False` for false and `1` or `True` for true, unless otherwise stated. (Important exception: the Boolean operations `or` and `and` always return one of their operands.)
These are the Boolean operations, ordered by ascending priority:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x or y</td>
<td>if x is false, then y, else x</td>
<td>(1)</td>
</tr>
<tr>
<td>x and y</td>
<td>if x is false, then x, else y</td>
<td>(2)</td>
</tr>
<tr>
<td>not x</td>
<td>if x is false, then True, else False</td>
<td>(3)</td>
</tr>
</tbody>
</table>

Notes:

1. This is a short-circuit operator, so it only evaluates the second argument if the first one is False.
2. This is a short-circuit operator, so it only evaluates the second argument if the first one is True.
3. not has a lower priority than non-Boolean operators, so not a == b is interpreted as not (a == b), and a == not b is a syntax error.
5.3. Comparisons

Comparison operations are supported by all objects. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example, \( x < y \leq z \) is equivalent to \( x < y \) and \( y \leq z \), except that \( y \) is evaluated only once (but in both cases \( z \) is not evaluated at all when \( x < y \) is found to be false).

This table summarizes the comparison operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>strictly less than</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>strictly greater than</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
<td>(1)</td>
</tr>
<tr>
<td>is</td>
<td>object identity</td>
<td></td>
</tr>
<tr>
<td>is not</td>
<td>negated object identity</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. \(!=\) can also be written \(<\>\), but this is an obsolete usage kept for backwards compatibility only. New code should always use \(!=\).

Objects of different types, except different numeric types and different string types, never compare equal; such objects are ordered consistently but arbitrarily (so that sorting a heterogeneous array yields a consistent result). Furthermore, some types (for example, file objects) support only a degenerate notion of comparison where any two objects of that type are unequal. Again, such objects are ordered arbitrarily but consistently. The \(<, \leq, >, \geq\) operators will
raise a `TypeError` exception when any operand is a complex number.

Non-identical instances of a class normally compare as non-equal unless the class defines the `__eq__()` method or the `__cmp__()` method.

Instances of a class cannot be ordered with respect to other instances of the same class, or other types of object, unless the class defines either enough of the rich comparison methods (`__lt__()`, `__le__()`, `__gt__()`, and `__ge__()`) or the `__cmp__()` method.

**CPython implementation detail:** Objects of different types except numbers are ordered by their type names; objects of the same types that don't support proper comparison are ordered by their address.

Two more operations with the same syntactic priority, `in` and `not in`, are supported only by sequence types (below).
There are four distinct numeric types: plain integers, long integers, floating point numbers, and complex numbers. In addition, Booleans are a subtype of plain integers. Plain integers (also just called integers) are implemented using long in C, which gives them at least 32 bits of precision (sys.maxint is always set to the maximum plain integer value for the current platform, the minimum value is -sys.maxint - 1). Long integers have unlimited precision. Floating point numbers are usually implemented using double in C; information about the precision and internal representation of floating point numbers for the machine on which your program is running is available in sys.float_info. Complex numbers have a real and imaginary part, which are each a floating point number. To extract these parts from a complex number z, use z.real and z.imag. (The standard library includes additional numeric types, fractions that hold rationals, and decimal that hold floating-point numbers with user-definable precision.)

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including binary, hex, and octal numbers) yield plain integers unless the value they denote is too large to be represented as a plain integer, in which case they yield a long integer. Integer literals with an 'L' or 'l' suffix yield long integers ('L' is preferred because 1l looks too much like eleven!). Numeric literals containing a decimal point or an exponent sign yield floating point numbers. Appending 'j' or 'J' to a numeric literal yields an imaginary number (a complex number with a zero real part) which you can add to an integer or float to get a complex number with real and imaginary parts.
Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the “narrower” type is widened to that of the other, where plain integer is narrower than long integer is narrower than floating point is narrower than complex. Comparisons between numbers of mixed type use the same rule. [2] The constructors `int()`, `long()`, `float()`, and `complex()` can be used to produce numbers of a specific type.

All built-in numeric types support the following operations. See The power operator and later sections for the operators’ priorities.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x + y</td>
<td>sum of x and y</td>
<td></td>
</tr>
<tr>
<td>x - y</td>
<td>difference of x and y</td>
<td></td>
</tr>
<tr>
<td>x * y</td>
<td>product of x and y</td>
<td></td>
</tr>
<tr>
<td>x / y</td>
<td>quotient of x and y</td>
<td>(1)</td>
</tr>
<tr>
<td>x // y</td>
<td>(floored) quotient of x and y</td>
<td>(4)(5)</td>
</tr>
<tr>
<td>x % y</td>
<td>remainder of x // y</td>
<td>(4)</td>
</tr>
<tr>
<td>-x</td>
<td>x negated</td>
<td></td>
</tr>
<tr>
<td>+x</td>
<td>x unchanged</td>
<td></td>
</tr>
<tr>
<td>abs(x)</td>
<td>absolute value or magnitude of x</td>
<td>(3)</td>
</tr>
<tr>
<td>int(x)</td>
<td>x converted to integer</td>
<td>(2)</td>
</tr>
<tr>
<td>long(x)</td>
<td>x converted to long integer</td>
<td>(2)</td>
</tr>
<tr>
<td>float(x)</td>
<td>x converted to floating point</td>
<td>(6)</td>
</tr>
<tr>
<td>complex(re,im)</td>
<td>a complex number with real part re, imaginary part im. im defaults to zero.</td>
<td></td>
</tr>
<tr>
<td>c.conjugate()</td>
<td>conjugate of the complex number c. (Identity on real numbers)</td>
<td></td>
</tr>
<tr>
<td>divmod(x, y)</td>
<td>the pair (x // y, x % y)</td>
<td>(3)(4)</td>
</tr>
<tr>
<td>pow(x, y)</td>
<td>x to the power y</td>
<td>(3)(7)</td>
</tr>
<tr>
<td>x ** y</td>
<td>x to the power y</td>
<td>(7)</td>
</tr>
</tbody>
</table>

Notes:
1. For (plain or long) integer division, the result is an integer. The result is always rounded towards minus infinity: 1/2 is 0, (-1)/2 is -1, 1/(-2) is -1, and (-1)/(-2) is 0. Note that the result is a long integer if either operand is a long integer, regardless of the numeric value.

2. Conversion from floats using int() or long() truncates toward zero like the related function, math.trunc(). Use the function math.floor() to round downward and math.ceil() to round upward.

3. See Built-in Functions for a full description.

4. Deprecated since version 2.3: The floor division operator, the modulo operator, and the divmod() function are no longer defined for complex numbers. Instead, convert to a floating point number using the abs() function if appropriate.

5. Also referred to as integer division. The resultant value is a whole integer, though the result's type is not necessarily int.

6. float also accepts the strings “nan” and “inf” with an optional prefix “+” or “-” for Not a Number (NaN) and positive or negative infinity.

   New in version 2.6.

7. Python defines pow(0, 0) and 0 ** 0 to be 1, as is common for programming languages.

All numbers.Real types (int, long, and float) also include the following operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
</table>

All numbers.Real types (int, long, and float) also include the following operations:
**5.4.1. Bitwise Operations on Integer Types**

Bitwise operations only make sense for integers. Negative numbers are treated as their 2’s complement value (this assumes a sufficiently large number of bits that no overflow occurs during the operation).

The priorities of the binary bitwise operations are all lower than the numeric operations and higher than the comparisons; the unary operation ~ has the same priority as the other unary numeric operations (+ and -).

This table lists the bitwise operations sorted in ascending priority:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>bitwise or of x and y</td>
</tr>
<tr>
<td>x ^ y</td>
<td>bitwise exclusive or of x and y</td>
<td></td>
</tr>
<tr>
<td>x &amp; y</td>
<td>bitwise and of x and y</td>
<td></td>
</tr>
<tr>
<td>x &lt;&lt; n</td>
<td>x shifted left by n bits</td>
<td>(1)(2)</td>
</tr>
<tr>
<td>x &gt;&gt; n</td>
<td>x shifted right by n bits</td>
<td>(1)(3)</td>
</tr>
<tr>
<td>~x</td>
<td>the bits of x inverted</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. Negative shift counts are illegal and cause a ValueError to be raised.
2. A left shift by n bits is equivalent to multiplication by pow(2, n).

   A long integer is returned if the result exceeds the range of plain
integers.
3. A right shift by $n$ bits is equivalent to division by $\text{pow}(2, n)$.

## 5.4.2. Additional Methods on Integer Types

The integer types implement the `numbers.Integral` abstract base class. In addition, they provide one more method:

```python
def bit_length(self):
    s = bin(self)  # binary representation: bin(-37) -
    s = s.lstrip('-0b')  # remove leading zeros and minus sig
    return len(s)  # len('100101') --> 6
```

`int.bit_length()`

`long.bit_length()`

Return the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros:

```python
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

More precisely, if $x$ is nonzero, then $x$.bit_length() is the unique positive integer $k$ such that $2^{(k-1)} \leq \text{abs}(x) < 2^k$. Equivalently, when $\text{abs}(x)$ is small enough to have a correctly rounded logarithm, then $k = 1 + \text{int}(\log(\text{abs}(x), 2))$. If $x$ is zero, then $x$.bit_length() returns 0.

Equivalent to:

```
New in version 2.7.
```

## 5.4.3. Additional Methods on Float
The float type implements the `numbers.Real` abstract base class. float also has the following additional methods.

**float.**`as_integer_ratio()`

Return a pair of integers whose ratio is exactly equal to the original float and with a positive denominator. Raises `OverflowError` on infinities and a `ValueError` on NaNs.

_New in version 2.6._

**float.**`is_integer()`

Return True if the float instance is finite with integral value, and False otherwise:

>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False

_New in version 2.6._

Two methods support conversion to and from hexadecimal strings. Since Python’s floats are stored internally as binary numbers, converting a float to or from a _decimal_ string usually involves a small rounding error. In contrast, hexadecimal strings allow exact representation and specification of floating-point numbers. This can be useful when debugging, and in numerical work.

**float.**`hex()`

Return a representation of a floating-point number as a hexadecimal string. For finite floating-point numbers, this representation will always include a leading 0x and a trailing p and exponent.

_New in version 2.6._
float.fromhex(s)

Class method to return the float represented by a hexadecimal string s. The string s may have leading and trailing whitespace.

*New in version 2.6.*

Note that `float.hex()` is an instance method, while `float.fromhex()` is a class method.

A hexadecimal string takes the form:

```
[sign] ['0x'] integer ['.' fraction] ['p' exponent]
```

where the optional `sign` may by either `+` or `-`, `integer` and `fraction` are strings of hexadecimal digits, and `exponent` is a decimal integer with an optional leading sign. Case is not significant, and there must be at least one hexadecimal digit in either the integer or the fraction. This syntax is similar to the syntax specified in section 6.4.4.2 of the C99 standard, and also to the syntax used in Java 1.5 onwards. In particular, the output of `float.hex()` is usable as a hexadecimal floating-point literal in C or Java code, and hexadecimal strings produced by C's `%a` format character or Java's `Double.toHexString` are accepted by `float.fromhex()`.

Note that the exponent is written in decimal rather than hexadecimal, and that it gives the power of 2 by which to multiply the coefficient. For example, the hexadecimal string `0x3.a7p10` represents the floating-point number \((3 + 10./16 + 7./16**2) * 2.0**10\), or 3740.0:

```
>>> float.fromhex('0x3.a7p10')
3740.0
```

Applying the reverse conversion to 3740.0 gives a different hexadecimal string representing the same number:
```python
>>> float.hex(3740.0)
'0x1.d38000000000p+11'
```
5.5. Iterator Types

New in version 2.2.

Python supports a concept of iteration over containers. This is implemented using two distinct methods; these are used to allow user-defined classes to support iteration. Sequences, described below in more detail, always support the iteration methods.

One method needs to be defined for container objects to provide iteration support:

```python
class container:
    def __iter__(self):
        # Return an iterator object. The object is required to support
        # the iterator protocol described below. If a container supports
        # different types of iteration, additional methods can be provided
        # to specifically request iterators for those iteration types. (An
        # example of an object supporting multiple forms of iteration would
        # be a tree structure which supports both breadth-first and depth-
        # first traversal.) This method corresponds to the \texttt{tp_iter}
        # slot of the type structure for Python objects in the Python/C API.
```

The iterator objects themselves are required to support the following two methods, which together form the iterator protocol:

```python
class iterator:
    def __iter__(self):
        # Return the iterator object itself. This is required to allow both
        # containers and iterators to be used with the \texttt{for} and \texttt{in}
        # statements. This method corresponds to the \texttt{tp_iter}
        # slot of the type structure for Python objects in the Python/C API.

    def next(self):
        # Return the next item from the container. If there are no further
        # items, raise the \texttt{StopIteration} exception. This method
```
corresponds to the `tp_iternext` slot of the type structure for Python objects in the Python/C API.

Python defines several iterator objects to support iteration over general and specific sequence types, dictionaries, and other more specialized forms. The specific types are not important beyond their implementation of the iterator protocol.

The intention of the protocol is that once an iterator’s `next()` method raises `StopIteration`, it will continue to do so on subsequent calls. Implementations that do not obey this property are deemed broken. (This constraint was added in Python 2.3; in Python 2.2, various iterators are broken according to this rule.)

5.5.1. Generator Types

Python’s `generators` provide a convenient way to implement the iterator protocol. If a container object’s `__iter__()` method is implemented as a generator, it will automatically return an iterator object (technically, a generator object) supplying the `__iter__()` and `next()` methods. More information about generators can be found in the documentation for the yield expression.
5.6. Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange

There are seven sequence types: strings, Unicode strings, lists, tuples, bytearrays, buffers, and xrange objects.

For other containers see the built in dict and set classes, and the collections module.

String literals are written in single or double quotes: 'xyzzy', "frobozz". See String literals for more about string literals. Unicode strings are much like strings, but are specified in the syntax using a preceding \u' character: u'abc', u"def". In addition to the functionality described here, there are also string-specific methods described in the String Methods section. Lists are constructed with square brackets, separating items with commas: [a, b, c]. Tuples are constructed by the comma operator (not within square brackets), with or without enclosing parentheses, but an empty tuple must have the enclosing parentheses, such as a, b, c or (). A single item tuple must have a trailing comma, such as (d,).

Bytearray objects are created with the built-in function bytearray().

Buffer objects are not directly supported by Python syntax, but can be created by calling the built-in function buffer(). They don’t support concatenation or repetition.

Objects of type xrange are similar to buffers in that there is no specific syntax to create them, but they are created using the xrange() function. They don’t support slicing, concatenation or repetition, and using in, not in, min() or max() on them is inefficient.
Most sequence types support the following operations. The `in` and `not in` operations have the same priorities as the comparison operations. The `+` and `*` operations have the same priority as the corresponding numeric operations. [3] Additional methods are provided for **Mutable Sequence Types**.

This table lists the sequence operations sorted in ascending priority. In the table, `s` and `t` are sequences of the same type; `n`, `i` and `j` are integers:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x in s</code></td>
<td>True if an item of <code>s</code> is equal to <code>x</code>, else <code>False</code></td>
<td>(1)</td>
</tr>
<tr>
<td><code>x not in s</code></td>
<td><code>False</code> if an item of <code>s</code> is equal to <code>x</code>, else <code>True</code></td>
<td>(1)</td>
</tr>
<tr>
<td><code>s + t</code></td>
<td>the concatenation of <code>s</code> and <code>t</code></td>
<td>(6)</td>
</tr>
<tr>
<td><code>s * n</code>, <code>n * s</code></td>
<td>equivalent to adding <code>s</code> to itself <code>n</code> times</td>
<td>(2)</td>
</tr>
<tr>
<td><code>s[i]</code></td>
<td><code>i</code>th item of <code>s</code>, origin 0</td>
<td>(3)</td>
</tr>
<tr>
<td><code>s[i:j]</code></td>
<td>slice of <code>s</code> from <code>i</code> to <code>j</code></td>
<td>(3)(4)</td>
</tr>
<tr>
<td><code>s[i:j:k]</code></td>
<td>slice of <code>s</code> from <code>i</code> to <code>j</code> with step <code>k</code></td>
<td>(3)(5)</td>
</tr>
<tr>
<td><code>len(s)</code></td>
<td>length of <code>s</code></td>
<td></td>
</tr>
<tr>
<td><code>min(s)</code></td>
<td>smallest item of <code>s</code></td>
<td></td>
</tr>
<tr>
<td><code>max(s)</code></td>
<td>largest item of <code>s</code></td>
<td></td>
</tr>
<tr>
<td><code>s.index(x)</code></td>
<td>index of the first occurrence of <code>x</code> in <code>s</code></td>
<td></td>
</tr>
<tr>
<td><code>s.count(x)</code></td>
<td>total number of occurrences of <code>x</code> in <code>s</code></td>
<td></td>
</tr>
</tbody>
</table>

Sequence types also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length. (For full details see [Comparisons in the...](#))
language reference.)

Notes:

1. When `s` is a string or Unicode string object the `in` and `not in` operations act like a substring test. In Python versions before 2.3, `x` had to be a string of length 1. In Python 2.3 and beyond, `x` may be a string of any length.

2. Values of `n` less than 0 are treated as 0 (which yields an empty sequence of the same type as `s`). Note that items in the sequence `s` are not copied; they are referenced multiple times. This often haunts new Python programmers; consider:

   ```python
   >>> lists = [[]] * 3
   >>> lists
   [[], [], []]
   >>> lists[0].append(3)
   >>> lists
   [[[3], [3], [3]]
   ```

   What has happened is that `[[ ]]` is a one-element list containing an empty list, so all three elements of `[[ ]] * 3` are references to this single empty list. Modifying any of the elements of `lists` modifies this single list. You can create a list of different lists this way:

   ```python
   >>> lists = [[] for i in range(3)]
   >>> lists[0].append(3)
   >>> lists[1].append(5)
   >>> lists[2].append(7)
   >>> lists
   [[[3], [5], [7]]
   ```

   Further explanation is available in the FAQ entry [How do I create a multidimensional list?](https://docs.python.org/2/faq/programming.html#how-do-i-create-a-multidimensional-list).

3. If `i` or `j` is negative, the index is relative to the end of the string:
len(s) + i or len(s) + j is substituted. But note that -0 is still 0.

4. The slice of s from i to j is defined as the sequence of items with index k such that \( i \leq k < j \). If \( i \) or \( j \) is greater than \( \text{len}(s) \), use \( \text{len}(s) \). If \( i \) is omitted or None, use 0. If \( j \) is omitted or None, use \( \text{len}(s) \). If \( i \) is greater than or equal to \( j \), the slice is empty.

5. The slice of s from i to j with step \( k \) is defined as the sequence of items with index \( x = i + n*k \) such that \( 0 \leq n < (j-i)/k \). In other words, the indices are \( i, i+k, i+2*k, i+3*k \) and so on, stopping when \( j \) is reached (but never including \( j \)). If \( i \) or \( j \) is greater than \( \text{len}(s) \), use \( \text{len}(s) \). If \( i \) or \( j \) are omitted or None, they become “end” values (which end depends on the sign of \( k \)). Note, \( k \) cannot be zero. If \( k \) is None, it is treated like 1.

6. **CPython implementation detail:** If \( s \) and \( t \) are both strings, some Python implementations such as CPython can usually perform an in-place optimization for assignments of the form \( s = s + t \) or \( s += t \). When applicable, this optimization makes quadratic run-time much less likely. This optimization is both version and implementation dependent. For performance sensitive code, it is preferable to use the \texttt{str.join()} method which assures consistent linear concatenation performance across versions and implementations.

*Changed in version 2.4:* Formerly, string concatenation never occurred in-place.

5.6.1. String Methods
Below are listed the string methods which both 8-bit strings and Unicode objects support. Some of them are also available on bytearray objects.

In addition, Python’s strings support the sequence type methods described in the Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange section. To output formatted strings use template strings or the % operator described in the String Formatting Operations section. Also, see the re module for string functions based on regular expressions.

**str.** *capitalize*

Return a copy of the string with its first character capitalized and the rest lowercased.

For 8-bit strings, this method is locale-dependent.

**str.** *center*(width[, fillchar])

Return centered in a string of length width. Padding is done using the specified fillchar (default is a space).

*Changed in version 2.4: Support for the fillchar argument.*

**str.** *count*(sub[, start[, end]])

Return the number of non-overlapping occurrences of substring sub in the range [start, end]. Optional arguments start and end are interpreted as in slice notation.

**str.** *decode*(encoding[, errors])

Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding. errors may be given to set a different error handling scheme. The default is 'strict', meaning that encoding errors raise UnicodeError. Other possible values are 'ignore', 'replace' and any other
name registered via `codecs.register_error()`, see section Codec Base Classes.

*New in version 2.2.*

*Changed in version 2.3:* Support for other error handling schemes added.

*Changed in version 2.7:* Support for keyword arguments added.

```
str.encode([encoding[, errors]])
```

Return an encoded version of the string. Default encoding is the current default string encoding. `errors` may be given to set a different error handling scheme. The default for `errors` is `'strict'`, meaning that encoding errors raise a `UnicodeError`. Other possible values are `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` and any other name registered via `codecs.register_error()`, see section Codec Base Classes. For a list of possible encodings, see section Standard Encodings.

*New in version 2.0.*

*Changed in version 2.3:* Support for `'xmlcharrefreplace'` and `'backslashreplace'` and other error handling schemes added.

*Changed in version 2.7:* Support for keyword arguments added.

```
str.endswith(suffix[, start[, end]])
```

Return `True` if the string ends with the specified `suffix`, otherwise return `False`. `suffix` can also be a tuple of suffixes to look for. With optional `start`, test beginning at that position. With optional `end`, stop comparing at that position.
Changed in version 2.5: Accept tuples as suffix.

str. **expandtabs**([**tabsize**])
Return a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every **tabsize** characters (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the string, the current column is set to zero and the string is examined character by character. If the character is a tab (\t), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the character is a newline (\n) or return (\r), it is copied and the current column is reset to zero. Any other character is copied unchanged and the current column is incremented by one regardless of how the character is represented when printed.

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01   012   0123  01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01   012  0123   01234'
```

str. **find**([**sub**[, **start**[, **end**]])
Return the lowest index in the string where substring **sub** is found within the slice **s[start:end]**. Optional arguments **start** and **end** are interpreted as in slice notation. Return -1 if **sub** is not found.

**Note:** The **find()** method should be used only if you need to know the position of **sub**. To check if **sub** is a substring or not, use the **in** operator:

```
>>> 'Py' in 'Python'
True
```
str. **`format(*args, **kwargs)`**

Perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces `{}`. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

```python
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

See **Format String Syntax** for a description of the various formatting options that can be specified in format strings.

This method of string formatting is the new standard in Python 3, and should be preferred to the `%` formatting described in **String Formatting Operations** in new code.

*New in version 2.6.*

str. **`index(sub[, start[, end]])`**

Like **`find()`**, but raise **`ValueError`** when the substring is not found.

str. **`isalnum()`**

Return true if all characters in the string are alphanumerical and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

str. **`isalpha()`**

Return true if all characters in the string are alphabetic and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.
str. **isdigit**()
Return true if all characters in the string are digits and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

str. **islower**()
Return true if all cased characters [4] in the string are lowercase and there is at least one cased character, false otherwise.

For 8-bit strings, this method is locale-dependent.

str. **isspace**()
Return true if there are only whitespace characters in the string and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

str. **istitle**()
Return true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return false otherwise.

For 8-bit strings, this method is locale-dependent.

str. **isupper**()
Return true if all cased characters [4] in the string are uppercase and there is at least one cased character, false otherwise.

For 8-bit strings, this method is locale-dependent.

str. **join**(iterable)
Return a string which is the concatenation of the strings in the iterable iterable. The separator between elements is the string providing this method.
str. `ljust(width[, fillchar])`

Return the string left justified in a string of length `width`. Padding is done using the specified `fillchar` (default is a space). The original string is returned if `width` is less than or equal to `len(s)`.

*Changed in version 2.4:* Support for the `fillchar` argument.

str. `lower()`

Return a copy of the string with all the cased characters [4] converted to lowercase.

For 8-bit strings, this method is locale-dependent.

str. `lstrip([chars])`

Return a copy of the string with leading characters removed. The `chars` argument is a string specifying the set of characters to be removed. If omitted or `None`, the `chars` argument defaults to removing whitespace. The `chars` argument is not a prefix; rather, all combinations of its values are stripped:

```python
>>> ' spacious '.lstrip()
'spacious'
>>> 'www.example.com'.lstrip('cmowz.')
'exmaple.com'
```

*Changed in version 2.2.2:* Support for the `chars` argument.

str. `partition(sep)`

Split the string at the first occurrence of `sep`, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

*New in version 2.5.*

str. `replace(old, new[, count])`
Return a copy of the string with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

```
str.rfind(sub[, start[, end]])
```
Return the highest index in the string where substring sub is found, such that sub is contained within s[start:end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

```
str.rindex(sub[, start[, end]])
```
Like rfind() but raises ValueError when the substring sub is not found.

```
str.rjust(width[, fillchar])
```
Return the string right justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than or equal to len(s).

*Changed in version 2.4:* Support for the fillchar argument.

```
str.rpartition(sep)
```
Split the string at the last occurrence of sep, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

*New in version 2.5.*

```
str.rsplit([sep[, maxsplit]])
```
Return a list of the words in the string, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done, the rightmost ones. If sep is not specified or None, any whitespace
string is a separator. Except for splitting from the right, \texttt{rsplit()} behaves like \texttt{split()} which is described in detail below.

\

\textit{New in version 2.4.}

\texttt{str.rstrip([chars])}

Return a copy of the string with trailing characters removed. The \textit{chars} argument is a string specifying the set of characters to be removed. If omitted or \texttt{None}, the \textit{chars} argument defaults to removing whitespace. The \textit{chars} argument is not a suffix; rather, all combinations of its values are stripped:

\begin{verbatim}
>>> '  spacious  '.rstrip()
  spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
\end{verbatim}

\textit{Changed in version 2.2.2: Support for the \textit{chars} argument.}

\texttt{str.split([sep[, maxsplit]])}

Return a list of the words in the string, using \textit{sep} as the delimiter string. If \textit{maxsplit} is given, at most \textit{maxsplit} splits are done (thus, the list will have at most \textit{maxsplit+1} elements). If \textit{maxsplit} is not specified or \texttt{-1}, then there is no limit on the number of splits (all possible splits are made).

If \textit{sep} is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, '1,,2'.split(',')) returns ['1', '', '2']). The \textit{sep} argument may consist of multiple characters (for example, '1<>2<>3'.split('<>') returns ['1', '2', '3']). Splitting an empty string with a specified separator returns ['',].

If \textit{sep} is not specified or is \texttt{None}, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single
separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a None separator returns [].

For example, ' 1 2 3 '.split() returns ['1', '2', '3'], and ' 1 2 3 '.split(None, 1) returns ['1', '2 3 '].

str.splitlines([keepends])
Return a list of the lines in the string, breaking at line boundaries. This method uses the universal newlines approach to splitting lines. Line breaks are not included in the resulting list unless keepends is given and true.

Python recognizes "\r", "\n", and "\r\n" as line boundaries for 8-bit strings.

For example:

```python
>>> 'ab c\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\nde fg\rkl\r\n'.splitlines(True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

Unlike split() when a delimiter string sep is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line:

```python
>>> ''.splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

For comparison, split(\n') gives:

```python
>>> ''.split(\n')
['']
```
unicode.splitlines([keepends])

Return a list of the lines in the string, like str.splitlines(). However, the Unicode method splits on the following line boundaries, which are a superset of the universal newlines recognized for 8-bit strings.

<table>
<thead>
<tr>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>Line Feed</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage Return</td>
</tr>
<tr>
<td>\r\n</td>
<td>Carriage Return + Line Feed</td>
</tr>
<tr>
<td>\v or \x0b</td>
<td>Line Tabulation</td>
</tr>
<tr>
<td>\f or \x0c</td>
<td>Form Feed</td>
</tr>
<tr>
<td>\x1c</td>
<td>File Separator</td>
</tr>
<tr>
<td>\x1d</td>
<td>Group Separator</td>
</tr>
<tr>
<td>\x1e</td>
<td>Record Separator</td>
</tr>
<tr>
<td>\x85</td>
<td>Next Line (C1 Control Code)</td>
</tr>
<tr>
<td>\u2028</td>
<td>Line Separator</td>
</tr>
<tr>
<td>\u2029</td>
<td>Paragraph Separator</td>
</tr>
</tbody>
</table>

*Changed in version 2.7:* \v and \f added to list of line boundaries.

str.startswith(prefix[, start[, end]])

Return True if string starts with the prefix, otherwise return False. prefix can also be a tuple of prefixes to look for. With optional start, test string beginning at that position. With optional end, stop comparing string at that position.

*Changed in version 2.5:* Accept tuples as prefix.
str. `strip([chars])`
Return a copy of the string with the leading and trailing characters removed. The `chars` argument is a string specifying the set of characters to be removed. If omitted or `None`, the `chars` argument defaults to removing whitespace. The `chars` argument is not a prefix or suffix; rather, all combinations of its values are stripped:

```python
>>> 'spacious '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'examp'
```

*Changed in version 2.2.2:* Support for the `chars` argument.

str. `swapcase()`
Return a copy of the string with uppercase characters converted to lowercase and vice versa.

For 8-bit strings, this method is locale-dependent.

str. `title()`
Return a titlecased version of the string where words start with an uppercase character and the remaining characters are lowercase.

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries, which may not be the desired result:

```python
>>> "they're bill's friends from the UK".title()
"They'RE Bill'S Friends From The Uk"
```

A workaround for apostrophes can be constructed using regular
expressions:

```python
>>> import re
def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                     lambda mo: mo.group(0)[0].upper() +
...                     mo.group(0)[1:].lower(),
...                     s)

>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

For 8-bit strings, this method is locale-dependent.

**str. `translate(table[, deletechars])`**

Return a copy of the string where all characters occurring in the optional argument `deletechars` are removed, and the remaining characters have been mapped through the given translation table, which must be a string of length 256.

You can use the `maketrans()` helper function in the `string` module to create a translation table. For string objects, set the `table` argument to `None` for translations that only delete characters:

```python
>>> 'read this short text'.translate(None, 'aeiou')
'rd ths shrt txt'
```

*New in version 2.6:* Support for a `None` `table` argument.

For Unicode objects, the `translate()` method does not accept the optional `deletechars` argument. Instead, it returns a copy of the `s` where all characters have been mapped through the given translation table which must be a mapping of Unicode ordinals to Unicode ordinals, Unicode strings or `None`. Unmapped characters are left untouched. Characters mapped to `None` are deleted. Note, a more flexible approach is to create a custom character mapping
codec using the `codecs` module (see `encodings.cp1251` for an example).

**str.upper()**

Return a copy of the string with all the cased characters [4] converted to uppercase. Note that `str.upper().isupper()` might be `False` if `s` contains uncased characters or if the Unicode category of the resulting character(s) is not “Lu” (Letter, uppercase), but e.g. “Lt” (Letter, titlecase).

For 8-bit strings, this method is locale-dependent.

**str.zfill(width)**

Return the numeric string left filled with zeros in a string of length `width`. A sign prefix is handled correctly. The original string is returned if `width` is less than or equal to `len(s)`.

*New in version 2.2.2.*

The following methods are present only on unicode objects:

**unicode.isnumeric()**

Return `True` if there are only numeric characters in `S`, `False` otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH.

**unicode.isdecimal()**

Return `True` if there are only decimal characters in `S`, `False` otherwise. Decimal characters include digit characters, and all characters that can be used to form decimal-radix numbers, e.g. U+0660, ARABIC-INDIC DIGIT ZERO.

### 5.6.2. String Formatting Operations
String and Unicode objects have one unique built-in operation: the % operator (modulo). This is also known as the string \textit{formatting} or \textit{interpolation} operator. Given \texttt{format \% values} (where \texttt{format} is a string or Unicode object), \% conversion specifications in \texttt{format} are replaced with zero or more elements of \texttt{values}. The effect is similar to the using \texttt{sprintf()} in the C language. If \texttt{format} is a Unicode object, or if any of the objects being converted using the \%s conversion are Unicode objects, the result will also be a Unicode object.

If \texttt{format} requires a single argument, \texttt{values} may be a single non-tuple object. \footnote{Other manual} Otherwise, \texttt{values} must be a tuple with exactly the number of items specified by the \texttt{format} string, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The \% character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, \texttt{(somename)}).
3. Conversion flags (optional), which affect the result of some conversion types.
4. Minimum field width (optional). If specified as an \*\* (asterisk), the actual width is read from the next element of the tuple in \texttt{values}, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a \. (dot) followed by the precision. If specified as \*\* (an asterisk), the actual width is read from the next element of the tuple in \texttt{values}, and the value to convert comes after the precision.
7. Conversion type.
When the right argument is a dictionary (or other mapping type), then the formats in the string must include a parenthesised mapping key into that dictionary inserted immediately after the '%' character. The mapping key selects the value to be formatted from the mapping. For example:

```python
>>> print '%(language)s has %(number)03d quote types.' % 
... {"language": "Python", "number": 2}
Python has 002 quote types.
```

In this case no * specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'#'</td>
<td>The value conversion will use the “alternate form” (where defined below).</td>
</tr>
<tr>
<td>'0'</td>
<td>The conversion will be zero padded for numeric values.</td>
</tr>
<tr>
<td>'-'</td>
<td>The converted value is left adjusted (overrides the '0' conversion if both are given).</td>
</tr>
<tr>
<td>' '</td>
<td>(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.</td>
</tr>
<tr>
<td>'+'</td>
<td>A sign character ('+' or '-' ) will precede the conversion (overrides a “space” flag).</td>
</tr>
</tbody>
</table>

A length modifier (h, l, or L) may be present, but is ignored as it is not necessary for Python – so e.g. %ld is identical to %d.

The conversion types are:

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>'d'</td>
<td>Signed integer decimal.</td>
<td></td>
</tr>
<tr>
<td>'i'</td>
<td>Signed integer decimal.</td>
<td></td>
</tr>
<tr>
<td>'o'</td>
<td>Signed octal value.</td>
<td>(1)</td>
</tr>
<tr>
<td>Format</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td><code>'u'</code></td>
<td>Obsolete type – it is identical to <code>'d'</code>.</td>
<td>(7)</td>
</tr>
<tr>
<td><code>'x'</code></td>
<td>Signed hexadecimal (lowercase).</td>
<td>(2)</td>
</tr>
<tr>
<td><code>'X'</code></td>
<td>Signed hexadecimal (uppercase).</td>
<td>(2)</td>
</tr>
<tr>
<td><code>'e'</code></td>
<td>Floating point exponential format (lowercase).</td>
<td>(3)</td>
</tr>
<tr>
<td><code>'E'</code></td>
<td>Floating point exponential format (uppercase).</td>
<td>(3)</td>
</tr>
<tr>
<td><code>'f'</code></td>
<td>Floating point decimal format.</td>
<td>(3)</td>
</tr>
<tr>
<td><code>'F'</code></td>
<td>Floating point decimal format.</td>
<td>(3)</td>
</tr>
<tr>
<td><code>'g'</code></td>
<td>Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.</td>
<td>(4)</td>
</tr>
<tr>
<td><code>'G'</code></td>
<td>Floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.</td>
<td>(4)</td>
</tr>
<tr>
<td><code>'c'</code></td>
<td>Single character (accepts integer or single character string).</td>
<td></td>
</tr>
<tr>
<td><code>'r'</code></td>
<td>String (converts any Python object using <code>repr()</code>).</td>
<td>(5)</td>
</tr>
<tr>
<td><code>'s'</code></td>
<td>String (converts any Python object using <code>str()</code>).</td>
<td>(6)</td>
</tr>
<tr>
<td><code>'%'</code></td>
<td>No argument is converted, results in a <code>'%'</code> character in the result.</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. The alternate form causes a leading zero (`'0'`) to be inserted between left-hand padding and the formatting of the number if the leading character of the result is not already a zero.

2. The alternate form causes a leading `0x` or `0X` (depending on whether the `'x'` or `'X'` format was used) to be inserted between left-hand padding and the formatting of the number if the leading character of the result is not already a zero.
3. The alternate form causes the result to always contain a decimal point, even if no digits follow it.

   The precision determines the number of digits after the decimal point and defaults to 6.

4. The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.

   The precision determines the number of significant digits before and after the decimal point and defaults to 6.

5. The %r conversion was added in Python 2.0.

   The precision determines the maximal number of characters used.

6. If the object or format provided is a `unicode` string, the resulting string will also be `unicode`.

   The precision determines the maximal number of characters used.

7. See PEP 237.

Since Python strings have an explicit length, `%s` conversions do not assume that `
\0` is the end of the string.

*Changed in version 2.7:* `%f` conversions for numbers whose absolute value is over 1e50 are no longer replaced by `%g` conversions.

Additional string operations are defined in standard modules `string` and `re`. 
5.6.3. XRange Type

The `xrange` type is an immutable sequence which is commonly used for looping. The advantage of the `xrange` type is that an `xrange` object will always take the same amount of memory, no matter the size of the range it represents. There are no consistent performance advantages.

XRange objects have very little behavior: they only support indexing, iteration, and the `len()` function.

5.6.4. Mutable Sequence Types

List and `bytearray` objects support additional operations that allow in-place modification of the object. Other mutable sequence types (when added to the language) should also support these operations. Strings and tuples are immutable sequence types: such objects cannot be modified once created. The following operations are defined on mutable sequence types (where `x` is an arbitrary object):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s[i] = x</code></td>
<td>item <code>i</code> of <code>s</code> is replaced by <code>x</code></td>
<td></td>
</tr>
<tr>
<td><code>s[i:j] = t</code></td>
<td>slice of <code>s</code> from <code>i</code> to <code>j</code> is replaced by the contents of the iterable <code>t</code></td>
<td></td>
</tr>
<tr>
<td><code>del s[i:j]</code></td>
<td>same as <code>s[i:j] = []</code></td>
<td></td>
</tr>
<tr>
<td><code>s[i:j:k] = t</code></td>
<td>the elements of <code>s[i:j:k]</code> are replaced by those of <code>t</code></td>
<td>(1)</td>
</tr>
<tr>
<td><code>del s[i:j:k]</code></td>
<td>removes the elements of <code>s[i:j:k]</code> from the list</td>
<td></td>
</tr>
<tr>
<td><code>s.append(x)</code></td>
<td>same as <code>s[len(s):len(s)] = [x]</code></td>
<td>(2)</td>
</tr>
<tr>
<td></td>
<td>for the most part the</td>
<td></td>
</tr>
</tbody>
</table>
s.extend(t) or s += t

same as
s[len(s):len(s)] = t

(3)

s *= n

updates s with its contents repeated n times

(11)

s.count(x)

return number of i's for which s[i] == x

s.index(x[, i[, j]])

return smallest k such that s[k] == x and i <= k < j

(4)

s.insert(i, x)

same as s[i:i] = [x]

(5)

s.pop([i])

same as x = s[i]; del s[i]; return x

(6)

s.remove(x)

same as del s[s.index(x)]

(4)

s.reverse()

reverses the items of s in place

(7)

s.sort([cmp[, key[, reverse]]]])

sort the items of s in place

(7)(8)(9)(10)

Notes:

1. t must have the same length as the slice it is replacing.

2. The C implementation of Python has historically accepted multiple parameters and implicitly joined them into a tuple; this no longer works in Python 2.0. Use of this misfeature has been deprecated since Python 1.4.

3. t can be any iterable object.

4. Raises ValueError when x is not found in s. When a negative index is passed as the second or third parameter to the index() method, the list length is added, as for slice indices. If it is still negative, it is truncated to zero, as for slice indices.
**Changed in version 2.3:** Previously, `index()` didn’t have arguments for specifying start and stop positions.

5. When a negative index is passed as the first parameter to the `insert()` method, the list length is added, as for slice indices. If it is still negative, it is truncated to zero, as for slice indices.

**Changed in version 2.3:** Previously, all negative indices were truncated to zero.

6. The `pop()` method’s optional argument `i` defaults to `-1`, so that by default the last item is removed and returned.

7. The `sort()` and `reverse()` methods modify the list in place for economy of space when sorting or reversing a large list. To remind you that they operate by side effect, they don’t return the sorted or reversed list.

8. The `sort()` method takes optional arguments for controlling the comparisons.

`cmp` specifies a custom comparison function of two arguments (list items) which should return a negative, zero or positive number depending on whether the first argument is considered smaller than, equal to, or larger than the second argument: `cmp=lambda x,y: cmp(x.lower(), y.lower())`. The default value is `None`.

`key` specifies a function of one argument that is used to extract a comparison key from each list element: `key=str.lower`. The default value is `None`.

`reverse` is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.
In general, the key and reverse conversion processes are much faster than specifying an equivalent cmp function. This is because cmp is called multiple times for each list element while key and reverse touch each element only once. Use functools.cmp_to_key() to convert an old-style cmp function to a key function.

*Changed in version 2.3:* Support for None as an equivalent to omitting cmp was added.

*Changed in version 2.4:* Support for key and reverse was added.

9. Starting with Python 2.3, the sort() method is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

10. **CPython implementation detail:** While a list is being sorted, the effect of attempting to mutate, or even inspect, the list is undefined. The C implementation of Python 2.3 and newer makes the list appear empty for the duration, and raises ValueError if it can detect that the list has been mutated during a sort.

11. The value n is an integer, or an object implementing __index__(). Zero and negative values of n clear the sequence. Items in the sequence are not copied; they are referenced multiple times, as explained for s * n under Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange.
A set object is an unordered collection of distinct hashable objects. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference. (For other containers see the built in dict, list, and tuple classes, and the collections module.)

New in version 2.4.

Like other collections, sets support x in set, len(set), and for x in set. Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

There are currently two built-in set types, set and frozenset. The set type is mutable — the contents can be changed using methods like add() and remove(). Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set. The frozenset type is immutable and hashable — its contents cannot be altered after it is created; it can therefore be used as a dictionary key or as an element of another set.

As of Python 2.7, non-empty sets (not frozensets) can be created by placing a comma-separated list of elements within braces, for example: {'jack', 'sjoerd'}, in addition to the set constructor.

The constructors for both classes work the same:

```python
class set(iterable)
class frozenset(iterable)
```

Return a new set or frozenset object whose elements are taken
from iterable. The elements of a set must be hashable. To represent sets of sets, the inner sets must be frozenset objects. If iterable is not specified, a new empty set is returned.

Instances of set and frozenset provide the following operations:

len(s)
Return the number of elements in set s (cardinality of s).

x in s
Test x for membership in s.

x not in s
Test x for non-membership in s.

isdisjoint(other)
Return True if the set has no elements in common with other. Sets are disjoint if and only if their intersection is the empty set.

New in version 2.6.

issubset(other)
set <= other
Test whether every element in the set is in other.

set < other
Test whether the set is a proper subset of other, that is, set <= other and set != other.

issuperset(other)
set >= other
Test whether every element in other is in the set.

set > other
Test whether the set is a proper superset of other, that is, set
other and set != other.

union(*others)
set | other | ...
    Return a new set with elements from the set and all others.

    Changed in version 2.6: Accepts multiple input iterables.

intersection(*others)
set & other & ...
    Return a new set with elements common to the set and all others.

    Changed in version 2.6: Accepts multiple input iterables.

difference(*others)
set - other - ...
    Return a new set with elements in the set that are not in the others.

    Changed in version 2.6: Accepts multiple input iterables.

symmetric_difference(other)
set ^ other
    Return a new set with elements in either the set or other but not both.

copy()
    Return a new set with a shallow copy of s.

Note, the non-operator versions of union(), intersection(), difference(), and symmetric_difference(), issubset(), and issuperset() methods will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes error-prone constructions like set('abc') & 'cbs' in favor of the more readable...
Both `set` and `frozenset` support set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other). A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal). A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

Instances of `set` are compared to instances of `frozenset` based on their members. For example, `set('abc') == frozenset('abc')` returns `True` and so does `set('abc') in set([frozenset('abc')])`.

The subset and equality comparisons do not generalize to a total ordering function. For example, any two non-empty disjoint sets are not equal and are not subsets of each other, so all of the following return `False`: `a<b`, `a==b`, or `a>b`. Accordingly, sets do not implement the `__cmp__()` method.

Since sets only define partial ordering (subset relationships), the output of the `list.sort()` method is undefined for lists of sets.

Set elements, like dictionary keys, must be `hashable`.

Binary operations that mix `set` instances with `frozenset` return the type of the first operand. For example: `frozenset('ab') | set('bc')` returns an instance of `frozenset`.

The following table lists operations available for `set` that do not apply to immutable instances of `frozenset`:

- `update(*others)`
- `set |== other | ...`
Update the set, adding elements from all others.

Changed in version 2.6: Accepts multiple input iterables.

```
intersection_update(*others)
set &= other & ...
```

Update the set, keeping only elements found in it and all others.

Changed in version 2.6:Accepts multiple input iterables.

```
difference_update(*others)
set -= other | ...
```

Update the set, removing elements found in others.

Changed in version 2.6: Accepts multiple input iterables.

```
symmetric_difference_update(other)
set ^= other
```

Update the set, keeping only elements found in either set, but not in both.

```
add(elem)
```
Add element `elem` to the set.

```
remove(elem)
```
Remove element `elem` from the set. Raises `KeyError` if `elem` is not contained in the set.

```
discard(elem)
```
Remove element `elem` from the set if it is present.

```
pop()
```
Remove and return an arbitrary element from the set. Raises `KeyError` if the set is empty.
clear()

Remove all elements from the set.

Note, the non-operator versions of the update(), intersection_update(), difference_update(), and symmetric_difference_update() methods will accept any iterable as an argument.

Note, the elem argument to the __contains__(), remove(), and discard() methods may be a set. To support searching for an equivalent frozenset, the elem set is temporarily mutated during the search and then restored. During the search, the elem set should not be read or mutated since it does not have a meaningful value.

See also:

Comparison to the built-in set types

Differences between the sets module and the built-in set types.
A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the dictionary. (For other containers see the built in list, set, and tuple classes, and the collections module.)

A dictionary's keys are almost arbitrary values. Values that are not hashable, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (such as 1 and 1.0) then they can be used interchangeably to index the same dictionary entry. (Note however, that since computers store floating-point numbers as approximations it is usually unwise to use them as dictionary keys.)

Dictionaries can be created by placing a comma-separated list of key: value pairs within braces, for example: {'jack': 4098, 'sjoerd': 4127} or {4098: 'jack', 4127: 'sjoerd'}, or by the dict constructor.

class dict(**kwarg)
class dict(mapping, **kwarg)
class dict(iterable, **kwarg)

Return a new dictionary initialized from an optional positional argument and a possibly empty set of keyword arguments.

If no positional argument is given, an empty dictionary is created. If a positional argument is given and it is a mapping object, a dictionary is created with the same key-value pairs as the mapping object. Otherwise, the positional argument must be an iterable object. Each item in the iterable must itself be an iterable
with exactly two objects. The first object of each item becomes a key in the new dictionary, and the second object the corresponding value. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument. If a key being added is already present, the value from the keyword argument replaces the value from the positional argument.

To illustrate, the following examples all return a dictionary equal to {"one": 1, "two": 2, "three": 3}:

```python
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

Providing keyword arguments as in the first example only works for keys that are valid Python identifiers. Otherwise, any valid keys can be used.

*New in version 2.2.*

*Changed in version 2.3:* Support for building a dictionary from keyword arguments added.

These are the operations that dictionaries support (and therefore, custom mapping types should support too):

`len(d)`

Return the number of items in the dictionary `d`. 
Return the item of \(d\) with key \(key\). Raises a \texttt{KeyError} if \(key\) is not in the map.

If a subclass of \texttt{dict} defines a method \texttt{__missing__()} and \(key\) is not present, the \(d[\text{key}]\) operation calls that method with the key \(key\) as argument. The \(d[\text{key}]\) operation then returns or raises whatever is returned or raised by the \texttt{__missing__()} call. No other operations or methods invoke \texttt{__missing__()}. If \texttt{__missing__()} is not defined, \texttt{KeyError} is raised. \texttt{__missing__()} must be a method; it cannot be an instance variable:

```python
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

The example above shows part of the implementation of \texttt{collections.Counter}. A different \texttt{__missing__()} method is used by \texttt{collections.defaultdict}.

\textit{New in version 2.5:} Recognition of \texttt{__missing__()} methods of \texttt{dict} subclasses.

\(d[\text{key}] = \texttt{value}\)

Set \(d[\text{key}]\) to \texttt{value}.

\texttt{del} \(d[\text{key}]\)

Remove \(d[\text{key}]\) from \(d\). Raises a \texttt{KeyError} if \(key\) is not in the map.
key in d
Return True if d has a key key, else False.

New in version 2.2.

key not in d
Equivalent to not key in d.

New in version 2.2.

iter(d)
Return an iterator over the keys of the dictionary. This is a shortcut for iterkeys().

clear()
Remove all items from the dictionary.

copy()
Return a shallow copy of the dictionary.

fromkeys(seq[, value])
Create a new dictionary with keys from seq and values set to value.

fromkeys() is a class method that returns a new dictionary. value defaults to None.

New in version 2.3.

get(key[, default])
Return the value for key if key is in the dictionary, else default. If default is not given, it defaults to None, so that this method never raises a KeyError.

has_key(key)
Test for the presence of `key` in the dictionary. `has_key()` is deprecated in favor of `key in d`.

**items()**

Return a copy of the dictionary's list of `(key, value)` pairs.

**CPython implementation detail:** Keys and values are listed in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions.

If `items()`, `keys()`, `values()`, `iteritems()`, `iterkeys()`, and `itervalues()` are called with no intervening modifications to the dictionary, the lists will directly correspond. This allows the creation of `(value, key)` pairs using `zip()`: `pairs = zip(d.values(), d.keys())`. The same relationship holds for the `iterkeys()` and `itervalues()` methods: `pairs = zip(d.itervalues(), d.iterkeys())` provides the same value for `pairs`. Another way to create the same list is `pairs = [(v, k) for (k, v) in d.iteritems()]`.

**iteritems()**

Return an iterator over the dictionary's `(key, value)` pairs. See the note for `dict.items()`.

Using `iteritems()` while adding or deleting entries in the dictionary may raise a `RuntimeError` or fail to iterate over all entries.

*New in version 2.2.*

**iterkeys()**

Return an iterator over the dictionary's keys. See the note for
dict.items().

Using iterkeys() while adding or deleting entries in the dictionary may raise a RuntimeError or fail to iterate over all entries.

*New in version 2.2.*

itervalues()

Return an iterator over the dictionary's values. See the note for dict.items().

Using itervalues() while adding or deleting entries in the dictionary may raise a RuntimeError or fail to iterate over all entries.

*New in version 2.2.*

keys()

Return a copy of the dictionary's list of keys. See the note for dict.items().

pop(key[, default])

If key is in the dictionary, remove it and return its value, else return default. If default is not given and key is not in the dictionary, a KeyError is raised.

*New in version 2.3.*

popitem()

Remove and return an arbitrary (key, value) pair from the dictionary.

popitem() is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling
popitem() raises a KeyError.

**setdefault**(key[, default])
If key is in the dictionary, return its value. If not, insert key with a value of default and return default. default defaults to None.

**update**(other)
Update the dictionary with the key/value pairs from other, overwriting existing keys. Return None.

    update() accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: d.update(red=1, blue=2).

    Changed in version 2.4: Allowed the argument to be an iterable of key/value pairs and allowed keyword arguments.

**values()**
Return a copy of the dictionary's list of values. See the note for **dict.items()**.

**viewitems()**
Return a new view of the dictionary's items ((key, value) pairs). See below for documentation of view objects.

    New in version 2.7.

**viewkeys()**
Return a new view of the dictionary's keys. See below for documentation of view objects.

    New in version 2.7.
**viewvalues()**

Return a new view of the dictionary's values. See below for documentation of view objects.

*New in version 2.7.*

Dictionaries compare equal if and only if they have the same (key, value) pairs.

### 5.8.1. Dictionary view objects

The objects returned by `dict.viewkeys()`, `dict.viewvalues()` and `dict.viewitems()` are *view objects*. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

Dictionary views can be iterated over to yield their respective data, and support membership tests:

**len(dictview)**

Return the number of entries in the dictionary.

**iter(dictview)**

Return an iterator over the keys, values or items (represented as tuples of `(key, value)`) in the dictionary.

Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond. This allows the creation of `(value, key)` pairs using `zip()`:

```python
pairs = zip(d.values(), d.keys())
```

Another way to create the same list is

```python
pairs = [(v, k) for (k, v) in d.items()]
```
Iterating views while adding or deleting entries in the dictionary may raise a `RuntimeError` or fail to iterate over all entries.

```
x in dictview
Return True if x is in the underlying dictionary's keys, values or items (in the latter case, x should be a (key, value) tuple).
```

Keys views are set-like since their entries are unique and hashable. If all values are hashable, so that (key, value) pairs are unique and hashable, then the items view is also set-like. (Values views are not treated as set-like since the entries are generally not unique.) Then these set operations are available (“other” refers either to another view or a set):

```
dictview & other
Return the intersection of the dictview and the other object as a new set.
dictview | other
Return the union of the dictview and the other object as a new set.
dictview - other
Return the difference between the dictview and the other object (all elements in dictview that aren’t in other) as a new set.
dictview ^ other
Return the symmetric difference (all elements either in dictview or other, but not in both) of the dictview and the other object as a new set.
```

An example of dictionary view usage:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.viewkeys()
>>> values = dishes.viewvalues()
```
```python
>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order
>>> list(keys)
['eggs', 'bacon', 'sausage', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['spam', 'bacon']

>>> # set operations
>>> keys & { 'eggs', 'bacon', 'salad'}
{'bacon'}
```
5.9. File Objects

File objects are implemented using C’s `stdio` package and can be created with the built-in `open()` function. File objects are also returned by some other built-in functions and methods, such as `os.popen()` and `os.fdopen()` and the `makefile()` method of socket objects. Temporary files can be created using the `tempfile` module, and high-level file operations such as copying, moving, and deleting files and directories can be achieved with the `shutil` module.

When a file operation fails for an I/O-related reason, the exception `IOError` is raised. This includes situations where the operation is not defined for some reason, like `seek()` on a tty device or writing a file opened for reading.

Files have the following methods:

`file.close()`
Close the file. A closed file cannot be read or written any more. Any operation which requires that the file be open will raise a `ValueError` after the file has been closed. Calling `close()` more than once is allowed.

As of Python 2.5, you can avoid having to call this method explicitly if you use the `with` statement. For example, the following code will automatically close `f` when the `with` block is exited:

```python
from __future__ import with_statement # This isn't required

with open("hello.txt") as f:
    for line in f:
        print line,
```
In older versions of Python, you would have needed to do this to get the same effect:

```python
def = open("hello.txt")
try:
    for line in f:
        print line,
finally:
    f.close()
```

**Note:** Not all “file-like” types in Python support use as a context manager for the `with` statement. If your code is intended to work with any file-like object, you can use the function `contextlib.closing()` instead of using the object directly.

**file.flush()**
Flush the internal buffer, like `stdio`'s `fflush()`. This may be a no-op on some file-like objects.

**Note:** `flush()` does not necessarily write the file’s data to disk. Use `flush()` followed by `os.fsync()` to ensure this behavior.

**file.fileno()**
Return the integer “file descriptor” that is used by the underlying implementation to request I/O operations from the operating system. This can be useful for other, lower level interfaces that use file descriptors, such as the `fcntl` module or `os.read()` and friends.

**Note:** File-like objects which do not have a real file descriptor should *not* provide this method!
Return `True` if the file is connected to a tty(-like) device, else `False`.

**Note:** If a file-like object is not associated with a real file, this method should *not* be implemented.

```python
file.next()
```

A file object is its own iterator, for example `iter(f)` returns `f` (unless `f` is closed). When a file is used as an iterator, typically in a `for` loop (for example, `for line in f: print line.strip()`), the `next()` method is called repeatedly. This method returns the next input line, or raises `StopIteration` when EOF is hit when the file is open for reading (behavior is undefined when the file is open for writing). In order to make a `for` loop the most efficient way of looping over the lines of a file (a very common operation), the `next()` method uses a hidden read-ahead buffer. As a consequence of using a read-ahead buffer, combining `next()` with other file methods (like `readline()`) does not work right. However, using `seek()` to reposition the file to an absolute position will flush the read-ahead buffer.

*New in version 2.3.*

```python
file.read([size])
```

Read at most `size` bytes from the file (less if the read hits EOF before obtaining `size` bytes). If the `size` argument is negative or omitted, read all data until EOF is reached. The bytes are returned as a string object. An empty string is returned when EOF is encountered immediately. (For certain files, like ttys, it makes sense to continue reading after an EOF is hit.) Note that this method may call the underlying C function `fread()` more than once in an effort to acquire as close to `size` bytes as possible. Also note that when in non-blocking mode, less data than was
requested may be returned, even if no size parameter was given.

Note: This function is simply a wrapper for the underlying fread() C function, and will behave the same in corner cases, such as whether the EOF value is cached.

file.readline([size])

Read one entire line from the file. A trailing newline character is kept in the string (but may be absent when a file ends with an incomplete line). If the size argument is present and non-negative, it is a maximum byte count (including the trailing newline) and an incomplete line may be returned. When size is not 0, an empty string is returned only when EOF is encountered immediately.

Note: Unlike stdio's fgets(), the returned string contains null characters ('\0') if they occurred in the input.

file.readlines([sizehint])

Read until EOF using readline() and return a list containing the lines thus read. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read. Objects implementing a file-like interface may choose to ignore sizehint if it cannot be implemented, or cannot be implemented efficiently.

file.xreadlines()

This method returns the same thing as iter(f).

New in version 2.1.

Deprecated since version 2.3: Use for line in file instead.
file.seek(offset[, whence])
Set the file's current position, like stdio's fseek(). The whence argument is optional and defaults to os.SEEK_SET or 0 (absolute file positioning); other values are os.SEEK_CUR or 1 (seek relative to the current position) and os.SEEK_END or 2 (seek relative to the file's end). There is no return value.

For example, f.seek(2, os.SEEK_CUR) advances the position by two and f.seek(-3, os.SEEK_END) sets the position to the third to last.

Note that if the file is opened for appending (mode 'a' or 'a+'), any seek() operations will be undone at the next write. If the file is only opened for writing in append mode (mode 'a'), this method is essentially a no-op, but it remains useful for files opened in append mode with reading enabled (mode 'a+'). If the file is opened in text mode (without 'b'), only offsets returned by tell() are legal. Use of other offsets causes undefined behavior.

Note that not all file objects are seekable.

*Changed in version 2.6:* Passing float values as offset has been deprecated.

file.tell()
Return the file's current position, like stdio's ftell().

**Note:** On Windows, tell() can return illegal values (after an fgets()) when reading files with Unix-style line-endings. Use binary mode ('rb') to circumvent this problem.

file.truncate([size])
Truncate the file's size. If the optional \texttt{size} argument is present, the file is truncated to (at most) that size. The size defaults to the current position. The current file position is not changed. Note that if a specified size exceeds the file's current size, the result is platform-dependent: possibilities include that the file may remain unchanged, increase to the specified size as if zero-filled, or increase to the specified size with undefined new content. Availability: Windows, many Unix variants.

\texttt{file.write(str)}

Write a string to the file. There is no return value. Due to buffering, the string may not actually show up in the file until the \texttt{flush()} or \texttt{close()} method is called.

\texttt{file.writelines(sequence)}

Write a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no return value. (The name is intended to match \texttt{readlines()}; \texttt{writelines()} does not add line separators.)

Files support the iterator protocol. Each iteration returns the same result as \texttt{readline()}, and iteration ends when the \texttt{readline()} method returns an empty string.

File objects also offer a number of other interesting attributes. These are not required for file-like objects, but should be implemented if they make sense for the particular object.

\texttt{file.closed}

bool indicating the current state of the file object. This is a read-only attribute; the \texttt{close()} method changes the value. It may not be available on all file-like objects.

\texttt{file.encoding}

The encoding that this file uses. When Unicode strings are
written to a file, they will be converted to byte strings using this encoding. In addition, when the file is connected to a terminal, the attribute gives the encoding that the terminal is likely to use (that information might be incorrect if the user has misconfigured the terminal). The attribute is read-only and may not be present on all file-like objects. It may also be `None`, in which case the file uses the system default encoding for converting Unicode strings.

*New in version 2.3.*

**file.errors**

The Unicode error handler used along with the encoding.

*New in version 2.6.*

**file.mode**

The I/O mode for the file. If the file was created using the `open()` built-in function, this will be the value of the `mode` parameter. This is a read-only attribute and may not be present on all file-like objects.

**file.name**

If the file object was created using `open()`, the name of the file. Otherwise, some string that indicates the source of the file object, of the form `<...>`. This is a read-only attribute and may not be present on all file-like objects.

**file.newlines**

If Python was built with universal newlines enabled (the default) this read-only attribute exists, and for files opened in universal newline read mode it keeps track of the types of newlines encountered while reading the file. The values it can take are `'\r'`, `'\n'`, `'\r\n'`, `None` (unknown, no newlines read yet) or a tuple containing all the newline types seen, to indicate that multiple newline conventions were encountered. For files not
opened in universal newlines read mode the value of this attribute will be `None`.

**file.softspace**

Boolean that indicates whether a space character needs to be printed before another value when using the `print` statement. Classes that are trying to simulate a file object should also have a writable `softspace` attribute, which should be initialized to zero. This will be automatic for most classes implemented in Python (care may be needed for objects that override attribute access); types implemented in C will have to provide a writable `softspace` attribute.

**Note:** This attribute is not used to control the `print` statement, but to allow the implementation of `print` to keep track of its internal state.
5.10. memoryview type

New in version 2.7.

memoryview objects allow Python code to access the internal data of an object that supports the buffer protocol without copying. Memory is generally interpreted as simple bytes.

class memoryview(obj)

Create a memoryview that references obj. obj must support the buffer protocol. Built-in objects that support the buffer protocol include str and bytearray (but not unicode).

A memoryview has the notion of an element, which is the atomic memory unit handled by the originating object obj. For many simple types such as str and bytearray, an element is a single byte, but other third-party types may expose larger elements.

len(view) returns the total number of elements in the memoryview, view. The itemsize attribute will give you the number of bytes in a single element.

A memoryview supports slicing to expose its data. Taking a single index will return a single element as a str object. Full slicing will result in a subview:

```python
>>> v = memoryview('abcefg')
>>> v[1]
'b'
>>> v[-1]
'g'
>>> v[1:4]
<memory at 0x77ab28>
>>> v[1:4].tobytes()
'bce'
```
If the object the memoryview is over supports changing its data, the memoryview supports slice assignment:

```python
>>> data = bytearray('abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = 'z'
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = '123'
>>> data
bytearray(b'z123fg')
>>> v[2] = 'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot modify size of memoryview object
```

Notice how the size of the memoryview object cannot be changed.

`memoryview` has two methods:

**tobytes()**
Return the data in the buffer as a bytestring (an object of class `str`).

```python
>>> m = memoryview("abc")
>>> m.tobytes()
'abc'
```

**tolist()**
Return the data in the buffer as a list of integers.

```python
>>> memoryview("abc").tolist()
[97, 98, 99]
```

There are also several readonly attributes available:

**format**
A string containing the format (in `struct` module style) for each element in the view. This defaults to `'B'`, a simple bytestring.

**itemsize**

The size in bytes of each element of the memoryview.

**shape**

A tuple of integers the length of `ndim` giving the shape of the memory as an N-dimensional array.

**ndim**

An integer indicating how many dimensions of a multidimensional array the memory represents.

**strides**

A tuple of integers the length of `ndim` giving the size in bytes to access each element for each dimension of the array.

**readonly**

A bool indicating whether the memory is read only.
5.11. Context Manager Types

New in version 2.5.

Python's `with` statement supports the concept of a runtime context defined by a context manager. This is implemented using two separate methods that allow user-defined classes to define a runtime context that is entered before the statement body is executed and exited when the statement ends.

The *context management protocol* consists of a pair of methods that need to be provided for a context manager object to define a runtime context:

```python
class contextmanager:
    def __enter__(self):
        Enter the runtime context and return either this object or another object related to the runtime context. The value returned by this method is bound to the identifier in the `as` clause of the `with` statement.

        An example of a context manager that returns itself is a file object. File objects return themselves from `__enter__()` to allow `open()` to be used as the context expression in a `with` statement.

        An example of a context manager that returns a related object is the one returned by `decimal.localcontext()`. These managers set the active decimal context to a copy of the original decimal context and then return the copy. This allows changes to be made to the current decimal context in the body of the `with` statement without affecting code outside the `with` statement.

    def __exit__(self, exc_type, exc_val, exc_tb):
        Exit the runtime context and return a Boolean flag indicating if
```
any exception that occurred should be suppressed. If an exception occurred while executing the body of the with statement, the arguments contain the exception type, value and traceback information. Otherwise, all three arguments are None.

Returning a true value from this method will cause the with statement to suppress the exception and continue execution with the statement immediately following the with statement. Otherwise the exception continues propagating after this method has finished executing. Exceptions that occur during execution of this method will replace any exception that occurred in the body of the with statement.

The exception passed in should never be reraised explicitly - instead, this method should return a false value to indicate that the method completed successfully and does not want to suppress the raised exception. This allows context management code (such as contextlib.nested) to easily detect whether or not an __exit__() method has actually failed.

Python defines several context managers to support easy thread synchronisation, prompt closure of files or other objects, and simpler manipulation of the active decimal arithmetic context. The specific types are not treated specially beyond their implementation of the context management protocol. See the contextlib module for some examples.

Python's generators and the contextlib.contextmanager decorator provide a convenient way to implement these protocols. If a generator function is decorated with the contextlib.contextmanager decorator, it will return a context manager implementing the necessary __enter__() and __exit__() methods, rather than the iterator produced by an undecorated generator function.
Note that there is no specific slot for any of these methods in the type structure for Python objects in the Python/C API. Extension types wanting to define these methods must provide them as a normal Python accessible method. Compared to the overhead of setting up the runtime context, the overhead of a single class dictionary lookup is negligible.
5.12. Other Built-in Types

The interpreter supports several other kinds of objects. Most of these support only one or two operations.

5.12.1. Modules

The only special operation on a module is attribute access: m.name, where m is a module and name accesses a name defined in m’s symbol table. Module attributes can be assigned to. (Note that the import statement is not, strictly speaking, an operation on a module object; import foo does not require a module object named foo to exist, rather it requires an (external) definition for a module named foo somewhere.)

A special attribute of every module is __dict__. This is the dictionary containing the module’s symbol table. Modifying this dictionary will actually change the module’s symbol table, but direct assignment to the __dict__ attribute is not possible (you can write m.__dict__['a'] = 1, which defines m.a to be 1, but you can’t write m.__dict__ = {}). Modifying __dict__ directly is not recommended.

Modules built into the interpreter are written like this: <module 'sys' (built-in)>. If loaded from a file, they are written as <module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>.

5.12.2. Classes and Class Instances

See Objects, values and types and Class definitions for these.

5.12.3. Functions
Function objects are created by function definitions. The only operation on a function object is to call it: \texttt{func(argument-list)}.

There are really two flavors of function objects: built-in functions and user-defined functions. Both support the same operation (to call the function), but the implementation is different, hence the different object types.

See Function definitions for more information.

5.12.4. Methods

Methods are functions that are called using the attribute notation. There are two flavors: built-in methods (such as \texttt{append()} on lists) and class instance methods. Built-in methods are described with the types that support them.

The implementation adds two special read-only attributes to class instance methods: \texttt{m.im_self} is the object on which the method operates, and \texttt{m.im_func} is the function implementing the method. Calling \texttt{m(arg-1, arg-2, ..., arg-n)} is completely equivalent to calling \texttt{m.im_func(m.im_self, arg-1, arg-2, ..., arg-n)}.

Class instance methods are either \texttt{bound} or \texttt{unbound}, referring to whether the method was accessed through an instance or a class, respectively. When a method is unbound, its \texttt{im_self} attribute will be \texttt{None} and if called, an explicit \texttt{self} object must be passed as the first argument. In this case, \texttt{self} must be an instance of the unbound method’s class (or a subclass of that class), otherwise a \texttt{TypeError} is raised.

Like function objects, methods objects support getting arbitrary attributes. However, since method attributes are actually stored on the underlying function object (\texttt{meth.im_func}), setting method
attributes on either bound or unbound methods is disallowed. Attempting to set an attribute on a method results in an AttributeError being raised. In order to set a method attribute, you need to explicitly set it on the underlying function object:

```python
>>> class C:
...     def method(self):
...         pass
... >>> c = C()
... >>> c.method.whoami = 'my name is method'  # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'instancemethod' object has no attribute 'whoami'
>>> c.method.im_func.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

See The standard type hierarchy for more information.

### 5.12.5. Code Objects

Code objects are used by the implementation to represent “pseudo-compiled” executable Python code such as a function body. They differ from function objects because they don’t contain a reference to their global execution environment. Code objects are returned by the built-in compile() function and can be extracted from function objects through their func_code attribute. See also the code module.

A code object can be executed or evaluated by passing it (instead of a source string) to the exec statement or the built-in eval() function.

See The standard type hierarchy for more information.

### 5.12.6. Type Objects

Type objects represent the various object types. An object’s type is
accessed by the built-in function \texttt{type()}. There are no special operations on types. The standard module \texttt{types} defines names for all standard built-in types.

Types are written like this: \texttt{<type 'int'>}.

### 5.12.7. The Null Object

This object is returned by functions that don’t explicitly return a value. It supports no special operations. There is exactly one null object, named \texttt{None} (a built-in name).

It is written as \texttt{None}.

### 5.12.8. The Ellipsis Object

This object is used by extended slice notation (see \texttt{Slicings}). It supports no special operations. There is exactly one ellipsis object, named \texttt{Ellipsis} (a built-in name).

It is written as \texttt{Ellipsis}. When in a subscript, it can also be written as \texttt{...}, for example \texttt{seq[...]}.

### 5.12.9. The NotImplemented Object

This object is returned from comparisons and binary operations when they are asked to operate on types they don’t support. See \texttt{Comparisons} for more information.

It is written as \texttt{NotImplemented}.

### 5.12.10. Boolean Values

Boolean values are the two constant objects \texttt{False} and \texttt{True}. They
are used to represent truth values (although other values can also be considered false or true). In numeric contexts (for example when used as the argument to an arithmetic operator), they behave like the integers 0 and 1, respectively. The built-in function `bool()` can be used to convert any value to a Boolean, if the value can be interpreted as a truth value (see section Truth Value Testing above).

They are written as `False` and `True`, respectively.

### 5.12.11. Internal Objects

See The standard type hierarchy for this information. It describes stack frame objects, traceback objects, and slice objects.
5.13. Special Attributes

The implementation adds a few special read-only attributes to several object types, where they are relevant. Some of these are not reported by the `dir()` built-in function.

**object.** __dict__

A dictionary or other mapping object used to store an object’s (writable) attributes.

**object.** __methods__

*Deprecated since version 2.2:* Use the built-in function `dir()` to get a list of an object’s attributes. This attribute is no longer available.

**object.** __members__

*Deprecated since version 2.2:* Use the built-in function `dir()` to get a list of an object’s attributes. This attribute is no longer available.

**instance.** __class__

The class to which a class instance belongs.

**class.** __bases__

The tuple of base classes of a class object.

**definition.** __name__

The name of the class, type, function, method, descriptor, or generator instance.

The following attributes are only supported by *new-style classes*.

**class.** __mro__

This attribute is a tuple of classes that are considered when
looking for base classes during method resolution.

class.mro()

This method can be overridden by a metaclass to customize the method resolution order for its instances. It is called at class instantiation, and its result is stored in __mro__.

class.__subclasses__()  

Each new-style class keeps a list of weak references to its immediate subclasses. This method returns a list of all those references still alive. Example:

```python
>>> int.__subclasses__()
[<type 'bool'>]
```

Footnotes

[1] Additional information on these special methods may be found in the Python Reference Manual (Basic customization).

[2] As a consequence, the list [1, 2] is considered equal to [1.0, 2.0], and similarly for tuples.

[3] They must have since the parser can’t tell the type of the operands.

(1, 2, 3, 4) Cased characters are those with general category property being one of “Lu” (Letter, uppercase), “Ll” (Letter, lowercase), or “Lt” (Letter, titlecase).

[4] To format only a tuple you should therefore provide a singleton tuple whose only element is the tuple to be formatted.

The advantage of leaving the newline on is that returning an empty string is then an unambiguous EOF indication. It is also possible (in cases where it might matter, for example, if you want to make an exact copy of a file while scanning its lines) to tell whether the last line of a file ended in a newline or not (yes this happens!).

[5] [6]
6. Built-in Exceptions

Exceptions should be class objects. The exceptions are defined in the module `exceptions`. This module never needs to be imported explicitly: the exceptions are provided in the built-in namespace as well as the `exceptions` module.

For class exceptions, in a `try` statement with an `except` clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which it is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an “associated value” indicating the detailed cause of the error. This may be a string or a tuple containing several items of information (e.g., an error code and a string explaining the code). The associated value is the second argument to the `raise` statement. If the exception class is derived from the standard root class `BaseException`, the associated value is present as the exception instance's `args` attribute.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition “just like” the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be subclassed to define new exceptions; programmers are encouraged to derive new exceptions from the `Exception` class or one of its subclasses, and not from `BaseException`. More information on defining exceptions is available...
in the Python Tutorial under User-defined Exceptions.

The following exceptions are only used as base classes for other exceptions.

exception BaseException
   The base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes (for that, use Exception). If str() or unicode() is called on an instance of this class, the representation of the argument(s) to the instance are returned, or the empty string when there were no arguments.

   New in version 2.5.

   args
   The tuple of arguments given to the exception constructor. Some built-in exceptions (like IOError) expect a certain number of arguments and assign a special meaning to the elements of this tuple, while others are usually called only with a single string giving an error message.

exception Exception
   All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class.

   Changed in version 2.5: Changed to inherit from BaseException.

exception StandardError
   The base class for all built-in exceptions except StopIteration, GeneratorExit, KeyboardInterrupt and SystemExit. StandardError itself is derived from Exception.

exception ArithmeticError
   The base class for those built-in exceptions that are raised for
various arithmetic errors: OverflowError, ZeroDivisionError, FloatingPointError.

**exception BufferError**
Raised when a buffer related operation cannot be performed.

**exception LookupError**
The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: IndexError, KeyError. This can be raised directly by codecs.lookup().

**exception EnvironmentError**
The base class for exceptions that can occur outside the Python system: IOError, OSError. When exceptions of this type are created with a 2-tuple, the first item is available on the instance’s errno attribute (it is assumed to be an error number), and the second item is available on the strerror attribute (it is usually the associated error message). The tuple itself is also available on the args attribute.

*New in version 1.5.2.*

When an EnvironmentError exception is instantiated with a 3-tuple, the first two items are available as above, while the third item is available on the filename attribute. However, for backwards compatibility, the args attribute contains only a 2-tuple of the first two constructor arguments.

The filename attribute is None when this exception is created with other than 3 arguments. The errno and strerror attributes are also None when the instance was created with other than 2 or 3 arguments. In this last case, args contains the verbatim constructor arguments as a tuple.
The following exceptions are the exceptions that are actually raised.

**exception AssertionError**
Raised when an `assert` statement fails.

**exception AttributeError**
Raised when an attribute reference (see Attribute references) or assignment fails. (When an object does not support attribute references or attribute assignments at all, `TypeError` is raised.)

**exception EOFError**
Raised when one of the built-in functions (`input()` or `raw_input()`) hits an end-of-file condition (EOF) without reading any data. (N.B.: the `file.read()` and `file.readline()` methods return an empty string when they hit EOF.)

**exception FloatingPointError**
Raised when a floating point operation fails. This exception is always defined, but can only be raised when Python is configured with the `--with-fpectl` option, or the `WANT_SIGFPE_HANDLER` symbol is defined in the `pyconfig.h` file.

**exception GeneratorExit**
Raised when a generator's `close()` method is called. It directly inherits from `BaseException` instead of `StandardError` since it is technically not an error.

*New in version 2.5.*

*Changed in version 2.6:* Changed to inherit from `BaseException`.

**exception IOError**
Raised when an I/O operation (such as a `print` statement, the built-in `open()` function or a method of a file object) fails for an
I/O-related reason, e.g., “file not found” or “disk full”.

This class is derived from `EnvironmentError`. See the discussion above for more information on exception instance attributes.

*Changed in version 2.6:* Changed `socket.error` to use this as a base class.

**exception** `ImportError`

Raised when an `import` statement fails to find the module definition or when a `from ... import` fails to find a name that is to be imported.

**exception** `IndexError`

Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not a plain integer, `TypeError` is raised.)

**exception** `KeyError`

Raised when a mapping (dictionary) key is not found in the set of existing keys.

**exception** `KeyboardInterrupt`

Raised when the user hits the interrupt key (normally `Control-C` or `Delete`). During execution, a check for interrupts is made regularly. Interrupts typed when a built-in function `input()` or `raw_input()` is waiting for input also raise this exception. The exception inherits from `BaseException` so as to not be accidentally caught by code that catches `Exception` and thus prevent the interpreter from exiting.

*Changed in version 2.5:* Changed to inherit from `BaseException`.

**exception** `MemoryError`
Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory. Note that because of the underlying memory management architecture (C’s `malloc()` function), the interpreter may not always be able to completely recover from this situation; it nevertheless raises an exception so that a stack traceback can be printed, in case a run-away program was the cause.

**exception NameError**
Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.

**exception NotImplementedError**
This exception is derived from `RuntimeError`. In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method.

*New in version 1.5.2.*

**exception OSError**
This exception is derived from `EnvironmentError`. It is raised when a function returns a system-related error (not for illegal argument types or other incidental errors). The `errno` attribute is a numeric error code from `errno`, and the `strerror` attribute is the corresponding string, as would be printed by the C function `perror()`. See the module `errno`, which contains names for the error codes defined by the underlying operating system.

For exceptions that involve a file system path (such as `chdir()` or `unlink()`), the exception instance will contain a third attribute, `filename`, which is the file name passed to the function.
New in version 1.5.2.

**exception OverflowError**

Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for long integers (which would rather raise MemoryError than give up) and for most operations with plain integers, which return a long integer instead. Because of the lack of standardization of floating point exception handling in C, most floating point operations also aren’t checked.

**exception ReferenceError**

This exception is raised when a weak reference proxy, created by the `weakref.proxy()` function, is used to access an attribute of the referent after it has been garbage collected. For more information on weak references, see the `weakref` module.

New in version 2.2: Previously known as the `weakref.ReferenceError` exception.

**exception RuntimeError**

Raised when an error is detected that doesn’t fall in any of the other categories. The associated value is a string indicating what precisely went wrong.

**exception StopIteration**

Raised by an iterator’s `next()` method to signal that there are no further values. This is derived from `Exception` rather than `StandardError`, since this is not considered an error in its normal application.

New in version 2.2.

**exception SyntaxError**

Raised when the parser encounters a syntax error. This may occur in an `import` statement, in an `exec` statement, in a call to
the built-in function `eval()` or `input()`, or when reading the initial
script or standard input (also interactively).

Instances of this class have attributes `filename`, `lineno`, `offset`
and `text` for easier access to the details. `str()` of the exception
instance returns only the message.

`exception IndentationError`
Base class for syntax errors related to incorrect indentation. This
is a subclass of `SyntaxError`.

`exception TabError`
Raised when indentation contains an inconsistent use of tabs and
spaces. This is a subclass of `IndentationError`.

`exception SystemError`
Raised when the interpreter finds an internal error, but the
situation does not look so serious to cause it to abandon all hope.
The associated value is a string indicating what went wrong (in
low-level terms).

You should report this to the author or maintainer of your Python
interpreter. Be sure to report the version of the Python interpreter
(`sys.version`; it is also printed at the start of an interactive Python
session), the exact error message (the exception’s associated
value) and if possible the source of the program that triggered the
error.

`exception SystemExit`
This exception is raised by the `sys.exit()` function. When it is not
handled, the Python interpreter exits; no stack traceback is
printed. If the associated value is a plain integer, it specifies the
system exit status (passed to C’s `exit()` function); if it is `None`,
the exit status is zero; if it has another type (such as a string), the
object's value is printed and the exit status is one.

Instances have an attribute `code` which is set to the proposed exit status or error message (defaulting to `None`). Also, this exception derives directly from `BaseException` and not `StandardError`, since it is not technically an error.

A call to `sys.exit()` is translated into an exception so that clean-up handlers (finally clauses of try statements) can be executed, and so that a debugger can execute a script without running the risk of losing control. The `os._exit()` function can be used if it is absolutely positively necessary to exit immediately (for example, in the child process after a call to `os.fork()`).

The exception inherits from `BaseException` instead of `StandardError` or `Exception` so that it is not accidentally caught by code that catches `Exception`. This allows the exception to properly propagate up and cause the interpreter to exit.

*Changed in version 2.5:* Changed to inherit from `BaseException`.

**exception** `TypeError`

Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

**exception** `UnboundLocalError`

Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. This is a subclass of `NameError`.

*New in version 2.0.*

**exception** `UnicodeError`
Raised when a Unicode-related encoding or decoding error occurs. It is a subclass of \texttt{ValueError}.

\texttt{UnicodeError} has attributes that describe the encoding or decoding error. For example, \texttt{err.object[err.start:err.end]} gives the particular invalid input that the codec failed on.

- **encoding**
  The name of the encoding that raised the error.

- **reason**
  A string describing the specific codec error.

- **object**
  The object the codec was attempting to encode or decode.

- **start**
  The first index of invalid data in \texttt{object}.

- **end**
  The index after the last invalid data in \texttt{object}.

\textit{New in version 2.0.}

\textit{exception} \texttt{UnicodeEncodeError}

Raised when a Unicode-related error occurs during encoding. It is a subclass of \texttt{UnicodeError}.

\textit{New in version 2.3.}

\textit{exception} \texttt{UnicodeDecodeError}

Raised when a Unicode-related error occurs during decoding. It is a subclass of \texttt{UnicodeError}.

\textit{New in version 2.3.}
exception UnicodeTranslateError
   Raised when a Unicode-related error occurs during translating. It is a subclass of UnicodeError.

   New in version 2.3.

exception ValueError
   Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as IndexError.

exception VMSError
   Only available on VMS. Raised when a VMS-specific error occurs.

exception WindowsError
   Raised when a Windows-specific error occurs or when the error number does not correspond to an errno value. The winerror and strerror values are created from the return values of the GetLastError() and FormatMessage() functions from the Windows Platform API. The errno value maps the winerror value to corresponding errno.h values. This is a subclass of OSError.

   New in version 2.0.

   Changed in version 2.5: Previous versions put the GetLastError() codes into errno.

exception ZeroDivisionError
   Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

The following exceptions are used as warning categories; see the
warnings module for more information.

**exception** Warning  
Base class for warning categories.

**exception** UserWarning  
Base class for warnings generated by user code.

**exception** DeprecationWarning  
Base class for warnings about deprecated features.

**exception** PendingDeprecationWarning  
Base class for warnings about features which will be deprecated in the future.

**exception** SyntaxWarning  
Base class for warnings about dubious syntax.

**exception** RuntimeWarning  
Base class for warnings about dubious runtime behavior.

**exception** FutureWarning  
Base class for warnings about constructs that will change semantically in the future.

**exception** ImportWarning  
Base class for warnings about probable mistakes in module imports.

*New in version 2.5.*

**exception** UnicodeWarning  
Base class for warnings related to Unicode.

*New in version 2.5.*
The class hierarchy for built-in exceptions is:

```
BaseException
 ++-- SystemExit
 ++-- KeyboardInterrupt
 ++-- GeneratorExit
 ++-- Exception
       ++-- StopIteration
       ++-- StandardError
               ++-- BufferError
               |      ++-- ArithmeticError
               |          |      ++-- FloatingPointError
               |          |      ++-- OverflowError
               |          |      ++-- ZeroDivisionError
               |      ++-- AssertionError
               |      ++-- AttributeError
               |      ++-- EnvironmentError
               |      |      ++-- IOError
               |      |      ++-- OSError
               |      |          |      ++-- WindowsError (Windows)
               |      |          |      ++-- VMSError (VMS)
               |      ++-- EOFError
               |      ++-- ImportError
               |      ++-- LookupError
               |      |      ++-- IndexError
               |      |      ++-- KeyError
               |      ++-- MemoryError
               |      |      ++-- UnboundLocalError
               |      ++-- ReferenceError
               |      ++-- RuntimeError
               |      |      ++-- NotImplementedError
               |      ++-- SyntaxError
               |          |      ++-- IndentationError
               |          |      ++-- TabError
               |      ++-- SystemError
               |      ++-- TypeError
               |      ++-- ValueError
               |          |      ++-- UnicodeError
               |          |          |      ++-- UnicodeDecodeError
               |          |          |      ++-- UnicodeEncodeError
               |          |      ++-- UnicodeTranslateError
```
7. String Services

The modules described in this chapter provide a wide range of string manipulation operations.

In addition, Python’s built-in string classes support the sequence type methods described in the Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange section, and also the string-specific methods described in the String Methods section. To output formatted strings use template strings or the \% operator described in the String Formatting Operations section. Also, see the re module for string functions based on regular expressions.

- 7.1. string — Common string operations
  - 7.1.1. String constants
  - 7.1.2. Custom String Formatting
  - 7.1.3. Format String Syntax
    - 7.1.3.1. Format Specification Mini-Language
    - 7.1.3.2. Format examples
  - 7.1.4. Template strings
  - 7.1.5. String functions
  - 7.1.6. Deprecated string functions
- 7.2. re — Regular expression operations
  - 7.2.1. Regular Expression Syntax
  - 7.2.2. Module Contents
  - 7.2.3. Regular Expression Objects
  - 7.2.4. Match Objects
  - 7.2.5. Examples
    - 7.2.5.1. Checking For a Pair
    - 7.2.5.2. Simulating scanf()
    - 7.2.5.3. search() vs. match()
    - 7.2.5.4. Making a Phonebook
    - 7.2.5.5. Text Munging
- 7.2.5.6. Finding all Adverbs
- 7.2.5.7. Finding all Adverbs and their Positions
- 7.2.5.8. Raw String Notation
- 7.3. struct — Interpret strings as packed binary data
  - 7.3.1. Functions and Exceptions
  - 7.3.2. Format Strings
    - 7.3.2.1. Byte Order, Size, and Alignment
    - 7.3.2.2. Format Characters
    - 7.3.2.3. Examples
  - 7.3.3. Classes
- 7.4. difflib — Helpers for computing deltas
  - 7.4.1. SequenceMatcher Objects
  - 7.4.2. SequenceMatcher Examples
  - 7.4.3. Differ Objects
  - 7.4.4. Differ Example
  - 7.4.5. A command-line interface to difflib
- 7.5. StringIO — Read and write strings as files
- 7.6. cStringIO — Faster version of StringIO
- 7.7. textwrap — Text wrapping and filling
- 7.8. codecs — Codec registry and base classes
  - 7.8.1. Codec Base Classes
    - 7.8.1.1. Codec Objects
    - 7.8.1.2. IncrementalEncoder Objects
    - 7.8.1.3. IncrementalDecoder Objects
    - 7.8.1.4. StreamWriter Objects
    - 7.8.1.5. StreamReader Objects
    - 7.8.1.6. StreamReaderWriter Objects
    - 7.8.1.7. StreamRecoder Objects
  - 7.8.2. Encodings and Unicode
  - 7.8.3. Standard Encodings
  - 7.8.4. Python Specific Encodings
  - 7.8.5. encodings.idna — Internationalized Domain Names in Applications
- 7.8.6. `encodings.utf_8_sig` — UTF-8 codec with BOM signature
- 7.9. `unicodedata` — Unicode Database
- 7.10. `stringprep` — Internet String Preparation
- 7.11. `fpformat` — Floating point conversions
The `string` module contains a number of useful constants and classes, as well as some deprecated legacy functions that are also available as methods on strings. In addition, Python’s built-in string classes support the sequence type methods described in the `Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange` section, and also the string-specific methods described in the `String Methods` section. To output formatted strings use template strings or the `%` operator described in the `String Formatting Operations` section. Also, see the `re` module for string functions based on regular expressions.
7.1.1. String constants

The constants defined in this module are:

- **ascii_letters**: The concatenation of the `ascii_lowercase` and `ascii_uppercase` constants described below. This value is not locale-dependent.

- **ascii_lowercase**: The lowercase letters 'abcdefghijklmnopqrstuvwxyz'. This value is not locale-dependent and will not change.

- **ascii_uppercase**: The uppercase letters 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. This value is not locale-dependent and will not change.

- **digits**: The string '0123456789'.

- **hexdigits**: The string '0123456789abcdefABCDEF'.

- **letters**: The concatenation of the strings `lowercase` and `uppercase` described below. The specific value is locale-dependent, and will be updated when `locale.setlocale()` is called.

- **lowercase**: A string containing all the characters that are considered lowercase letters. On most systems this is the string 'abcdefghijklmnopqrstuvwxyz'. The specific value is locale-dependent, and will be updated when `locale.setlocale()` is called.
string. octdigits

The string '01234567'.

string. punctuation

String of ASCII characters which are considered punctuation characters in the C locale.

string. printable

String of characters which are considered printable. This is a combination of digits, letters, punctuation, and whitespace.

string. uppercase

A string containing all the characters that are considered uppercase letters. On most systems this is the string 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. The specific value is locale-dependent, and will be updated when locale.setlocale() is called.

string. whitespace

A string containing all characters that are considered whitespace. On most systems this includes the characters space, tab, linefeed, return, formfeed, and vertical tab.
7.1.2. Custom String Formatting

*New in version 2.6.*

The built-in str and unicode classes provide the ability to do complex variable substitutions and value formatting via the `str.format()` method described in PEP 3101. The `Formatter` class in the `string` module allows you to create and customize your own string formatting behaviors using the same implementation as the built-in `format()` method.

```python
class string.Formatter

    The `Formatter` class has the following public methods:

    `format(format_string, *args, **kwargs)`
    The primary API method. It takes a format string and an arbitrary set of positional and keyword arguments. It is just a wrapper that calls `vformat()`.

    `vformat(format_string, args, kwargs)`
    This function does the actual work of formatting. It is exposed as a separate function for cases where you want to pass in a predefined dictionary of arguments, rather than unpacking and repacking the dictionary as individual arguments using the `*args` and `**kwargs` syntax. `vformat()` does the work of breaking up the format string into character data and replacement fields. It calls the various methods described below.

    In addition, the `Formatter` defines a number of methods that are intended to be replaced by subclasses:

    `parse(format_string)`
```
Loop over the format_string and return an iterable of tuples
(literal_text, field_name, format_spec, conversion). This is
used by vformat() to break the string into either literal text, or
replacement fields.

The values in the tuple conceptually represent a span of literal
text followed by a single replacement field. If there is no literal
text (which can happen if two replacement fields occur
consecutively), then literal_text will be a zero-length string. If
there is no replacement field, then the values of field_name,
format_spec and conversion will be None.

get_field(field_name, args, kwargs)
Given field_name as returned by parse() (see above),
convert it to an object to be formatted. Returns a tuple (obj,
used_key). The default version takes strings of the form
defined in PEP 3101, such as “0[name]” or “label.title”. args
and kwargs are as passed in to vformat(). The return value
used_key has the same meaning as the key parameter to
get_value().

get_value(key, args, kwargs)
Retrieve a given field value. The key argument will be either
an integer or a string. If it is an integer, it represents the index
of the positional argument in args; if it is a string, then it
represents a named argument in kwargs.

The args parameter is set to the list of positional arguments to
vformat(), and the kwargs parameter is set to the dictionary
of keyword arguments.

For compound field names, these functions are only called for
the first component of the field name; Subsequent
components are handled through normal attribute and
indexing operations.

So for example, the field expression '0.name' would cause `get_value()` to be called with a key argument of 0. The name attribute will be looked up after `get_value()` returns by calling the built-in `getattr()` function.

If the index or keyword refers to an item that does not exist, then an `IndexError` or `KeyError` should be raised.

**check_unused_args(used_args, args, kwargs)**
Implement checking for unused arguments if desired. The arguments to this function is the set of all argument keys that were actually referred to in the format string (integers for positional arguments, and strings for named arguments), and a reference to the args and kwargs that was passed to vformat. The set of unused args can be calculated from these parameters. `check_unused_args()` is assumed to raise an exception if the check fails.

**format_field(value, format_spec)**
`format_field()` simply calls the global `format()` built-in. The method is provided so that subclasses can override it.

**convert_field(value, conversion)**
Converts the value (returned by `get_field()`) given a conversion type (as in the tuple returned by the `parse()` method). The default version understands 's' (str), 'r' (repr) and 'a' (ascii) conversion types.
7.1.3. Format String Syntax

The `str.format()` method and the `Formatter` class share the same syntax for format strings (although in the case of `Formatter`, subclasses can define their own format string syntax).

Format strings contain “replacement fields” surrounded by curly braces `{}`. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: `{{` and `}}`.

The grammar for a replacement field is as follows:

```
replacement_field ::= "{" [field_name] ["!" conversion] ["":] format_spec
field_name ::= arg_name ("." attribute_name | "]" element_index
arg_name ::= [identifier | integer]
attribute_name ::= identifier
element_index ::= integer | index_string
index_string ::= <any source character except "]"> +
conversion ::= "r" | "s"
format_spec ::= <described in the next section>
```

In less formal terms, the replacement field can start with a `field_name` that specifies the object whose value is to be formatted and inserted into the output instead of the replacement field. The `field_name` is optionally followed by a `conversion` field, which is preceded by an exclamation point `!!`, and a `format_spec`, which is preceded by a colon `:;`. These specify a non-default format for the replacement value.

See also the Format Specification Mini-Language section.

The `field_name` itself begins with an `arg_name` that is either a number or a keyword. If it’s a number, it refers to a positional
argument, and if it's a keyword, it refers to a named keyword argument. If the numerical arg_names in a format string are 0, 1, 2, ... in sequence, they can all be omitted (not just some) and the numbers 0, 1, 2, ... will be automatically inserted in that order. Because arg_name is not quote-delimited, it is not possible to specify arbitrary dictionary keys (e.g., the strings '10' or ':--]') within a format string. The arg_name can be followed by any number of index or attribute expressions. An expression of the form `.name` selects the named attribute using getattr(), while an expression of the form `[index]` does an index lookup using __getitem__().

*Changed in version 2.7:* The positional argument specifiers can be omitted, so `{}` `{}` is equivalent to `{0} {1}`.

Some simple format string examples:

```
"First, thou shalt count to {0}"  # References first positional
"Bring me a {}"                   # Implicitly references the first
"From {} to {}"                   # Same as "From {0} to {1}"
"My quest is {name}"             # References keyword argument
"Weight in tons {0.weight}"      # 'weight' attribute of first
"Units destroyed: {players[0]}"  # First element of keyword ar
```

The conversion field causes a type coercion before formatting. Normally, the job of formatting a value is done by the __format__() method of the value itself. However, in some cases it is desirable to force a type to be formatted as a string, overriding its own definition of formatting. By converting the value to a string before calling __format__(), the normal formatting logic is bypassed.

Two conversion flags are currently supported: '!s' which calls str() on the value, and '!r' which calls repr().

Some examples:
The *format_spec* field contains a specification of how the value should be presented, including such details as field width, alignment, padding, decimal precision and so on. Each value type can define its own “formatting mini-language” or interpretation of the *format_spec*.

Most built-in types support a common formatting mini-language, which is described in the next section.

A *format_spec* field can also include nested replacement fields within it. These nested replacement fields may contain a field name, conversion flag and format specification, but deeper nesting is not allowed. The replacement fields within the *format_spec* are substituted before the *format_spec* string is interpreted. This allows the formatting of a value to be dynamically specified.

See the Format examples section for some examples.

### 7.1.3.1. Format Specification Mini-Language

“Format specifications” are used within replacement fields contained within a format string to define how individual values are presented (see Format String Syntax). They can also be passed directly to the built-in *format()* function. Each formattable type may define how the format specification is to be interpreted.

Most built-in types implement the following options for format specifications, although some of the formatting options are only supported by the numeric types.

A general convention is that an empty format string (""") produces the same result as if you had called *str()* on the value. A non-empty format string typically modifies the result.
The general form of a **standard format specifier** is:

\[
\text{format_spec ::= \{[\text{fill]}\text{align}][\text{sign}][#][0][\text{width}][,][.\text{precision}]}\]

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fill</strong></td>
<td>&lt;any character&gt;</td>
</tr>
<tr>
<td><strong>align</strong></td>
<td>&quot;&lt;&quot;</td>
</tr>
<tr>
<td><strong>sign</strong></td>
<td>&quot;+&quot;</td>
</tr>
<tr>
<td><strong>width</strong></td>
<td>integer</td>
</tr>
<tr>
<td><strong>precision</strong></td>
<td>integer</td>
</tr>
<tr>
<td><strong>type</strong></td>
<td>&quot;b&quot;</td>
</tr>
</tbody>
</table>

If a valid **align** value is specified, it can be preceded by a **fill** character that can be any character and defaults to a space if omitted. It is not possible to use a literal curly brace ("{" or "}"") as the **fill** character when using the ```str.format()``` method. However, it is possible to insert a curly brace with a nested replacement field. This limitation doesn’t affect the ```format()``` function.

The meaning of the various alignment options is as follows:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'&lt;'</td>
<td>Forces the field to be left-aligned within the available space (this is the default for most objects).</td>
</tr>
<tr>
<td>'&gt;'</td>
<td>Forces the field to be right-aligned within the available space (this is the default for numbers).</td>
</tr>
<tr>
<td>'='</td>
<td>Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form ‘+00000120’. This alignment option is only valid for numeric types. It becomes the default when ‘0’ immediately precedes the field width.</td>
</tr>
<tr>
<td>'^'</td>
<td>Forces the field to be centered within the available space.</td>
</tr>
</tbody>
</table>

Note that unless a minimum field width is defined, the field width will always be the same size as the data to fill it, so that the alignment option has no meaning in this case.
The *sign* option is only valid for number types, and can be one of the following:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'+'</td>
<td>indicates that a sign should be used for both positive as well as negative numbers.</td>
</tr>
<tr>
<td>'-'</td>
<td>indicates that a sign should be used only for negative numbers (this is the default behavior).</td>
</tr>
<tr>
<td>space</td>
<td>indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.</td>
</tr>
</tbody>
</table>

The '*' option is only valid for integers, and only for binary, octal, or hexadecimal output. If present, it specifies that the output will be prefixed by '0b', '0o', or '0x', respectively.

The ',' option signals the use of a comma as a thousands separator. For a locale aware separator, use the 'n' integer presentation type instead.

*Changed in version 2.7: Added the ',' option* (see also PEP 378).

*width* is a decimal integer defining the minimum field width. If not specified, then the field width will be determined by the content.

When no explicit alignment is given, preceding the *width* field by a zero ('0') character enables sign-aware zero-padding for numeric types. This is equivalent to a *fill* character of '0' with an *alignment* type of '='.

The *precision* is a decimal number indicating how many digits should be displayed after the decimal point for a floating point value formatted with 'f' and 'F', or before and after the decimal point for a floating point value formatted with 'g' or 'G'. For non-number types the field indicates the maximum field size - in other words, how
many characters will be used from the field content. The *precision* is not allowed for integer values.

Finally, the *type* determines how the data should be presented.

The available string presentation types are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'s'</td>
<td>String format. This is the default type for strings and may be omitted.</td>
</tr>
<tr>
<td>None</td>
<td>The same as 's'.</td>
</tr>
</tbody>
</table>

The available integer presentation types are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'b'</td>
<td>Binary format. Outputs the number in base 2.</td>
</tr>
<tr>
<td>'c'</td>
<td>Character. Converts the integer to the corresponding unicode character before printing.</td>
</tr>
<tr>
<td>'d'</td>
<td>Decimal Integer. Outputs the number in base 10.</td>
</tr>
<tr>
<td>'o'</td>
<td>Octal format. Outputs the number in base 8.</td>
</tr>
<tr>
<td>'x'</td>
<td>Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9.</td>
</tr>
<tr>
<td>'X'</td>
<td>Hex format. Outputs the number in base 16, using upper-case letters for the digits above 9.</td>
</tr>
<tr>
<td>'n'</td>
<td>Number. This is the same as 'd', except that it uses the current locale setting to insert the appropriate number separator characters.</td>
</tr>
<tr>
<td>None</td>
<td>The same as 'd'.</td>
</tr>
</tbody>
</table>

In addition to the above presentation types, integers can be formatted with the floating point presentation types listed below (except 'n' and None). When doing so, `float()` is used to convert the integer to a floating point number before formatting.

The available presentation types for floating point and decimal
values are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'e'</td>
<td>Exponent notation. Prints the number in scientific notation using the letter 'e' to indicate the exponent. The default precision is 6.</td>
</tr>
<tr>
<td>'E'</td>
<td>Exponent notation. Same as 'e' except it uses an upper case 'E' as the separator character.</td>
</tr>
<tr>
<td>'f'</td>
<td>Fixed point. Displays the number as a fixed-point number. The default precision is 6.</td>
</tr>
<tr>
<td>'F'</td>
<td>Fixed point. Same as 'f'.</td>
</tr>
<tr>
<td>'g'</td>
<td>General format. For a given precision ( p &gt;= 1 ), this rounds the number to ( p ) significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude. The precise rules are as follows: suppose that the result formatted with presentation type 'e' and precision ( p-1 ) would have exponent ( \exp ). Then if (-4 &lt;= \exp &lt; p), the number is formatted with presentation type 'f' and precision ( p-1-\exp ). Otherwise, the number is formatted with presentation type 'e' and precision ( p-1 ). In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it. Positive and negative infinity, positive and negative zero, and nans, are formatted as inf, -inf, 0, -0 and nan respectively, regardless of the precision. A precision of 0 is treated as equivalent to a precision of 1. The default precision is 6.</td>
</tr>
</tbody>
</table>

General format. Same as 'g' except switches to 'E' if
The number gets too large. The representations of infinity and NaN are uppercased, too.

Number. This is the same as 'g', except that it uses the current locale setting to insert the appropriate number separator characters.

Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.

None The same as 'g'.

7.1.3.2. Format examples

This section contains examples of the `str.format()` syntax and comparison with the old `%`-formatting.

In most of the cases the syntax is similar to the old `%`-formatting, with the addition of the `{}` and with : used instead of %. For example, `%03.2f` can be translated to `{0:03.2f}`.

The new format syntax also supports new and different options, shown in the follow examples.

Accessing arguments by position:

```python
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 2.7+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2, 1, 0}'.format('*abc')  # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')  # arguments' indices can
'abracadabra'
```

Accessing arguments by name:
Accessing arguments’ attributes:

```python
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.0.'
```

```python
class Point(object):
    ...    def __init__(self, x, y):
    ...        self.x, self.y = x, y
    ...    def __str__(self):
    ...        return 'Point({self.x}, {self.y})'.format(self=self)

>>> str(Point(4, 2))
'Point(4, 2)'
```

Accessing arguments’ items:

```python
>>> coord = (3, 5)
>>> '{0[0]}; {0[1]}'.format(coord)
'X: 3; Y: 5'
```

Replacing `%s` and `%r`:

```python
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1')
'repr() shows quotes: 'test1'; str() doesn't: test1'
```

Aligning the text and specifying a width:

```python
>>> '{:<30}'.format('left aligned')
'left aligned'
>>> '{:>30}'.format('right aligned')
'right aligned'
>>> '{:^30}'.format('centered')
'centered'
```
Replacing ``%+f``, ``%-f``, and ``%f`` and specifying a sign:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
>>> '{:f}; {: f}'.format(3.14, -3.14)  # show a space for posi
'  3.140000; -3.140000'
>>> '{:-f}; {-:f}'.format(3.14, -3.14)  # show only the minus -
' -3.140000; -3.140000'
```

Replacing ``%x`` and ``%o`` and converting the value to different bases:

```
>>> # format also supports binary numbers
>>> "int: {0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}".format(42)
'int: 42; hex: 2a; oct: 52; bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}".format(42)
'int: 42; hex: 0x2a; oct: 0o52; bin: 0b101010'
```

Using the comma as a thousands separator:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

Expressing a percentage:

```
>>> points = 19.5
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 88.64%'
```

Using type-specific formatting:

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```
Nesting arguments and more complex examples:

```python
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>right'

>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_ , 16)
3232235521

>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print '{0:{width}{base}}'.format(num, base=base, width=width)
...         print
...
    5      5      5      101
    6      6      6      110
    7      7      7      111
    8      8      10      1000
    9      9      11      1001
   10      A      12      1010
   11      B      13      1011
```
7.1.4. Template strings

New in version 2.4.

Templates provide simpler string substitutions as described in PEP 292. Instead of the normal %-based substitutions, Templates support $-based substitutions, using the following rules:

- $$ is an escape; it is replaced with a single $.
- $identifier names a substitution placeholder matching a mapping key of "identifier". By default, "identifier" must spell a Python identifier. The first non-identifier character after the $ character terminates this placeholder specification.
- ${identifier} is equivalent to $identifier. It is required when valid identifier characters follow the placeholder but are not part of the placeholder, such as "${noun}ification".

Any other appearance of $ in the string will result in a ValueError being raised.

The string module provides a Template class that implements these rules. The methods of Template are:

```python
class string.Template(template)
    The constructor takes a single argument which is the template string.

    substitute(mapping[, **kws])
    Performs the template substitution, returning a new string. mapping is any dictionary-like object with keys that match the placeholders in the template. Alternatively, you can provide keyword arguments, where the keywords are the placeholders. When both mapping and kws are given and
there are duplicates, the placeholders from \textit{kws} take precedence.

\textbf{safe\_substitute}(\texttt{mapping}, **\texttt{kws})

Like \texttt{substitute()}, except that if placeholders are missing from \texttt{mapping} and \texttt{kws}, instead of raising a \texttt{KeyError} exception, the original placeholder will appear in the resulting string intact. Also, unlike with \texttt{substitute()}, any other appearances of the $ will simply return $ instead of raising \texttt{ValueError}.

While other exceptions may still occur, this method is called “safe” because substitutions always tries to return a usable string instead of raising an exception. In another sense, \texttt{safe\_substitute()} may be anything other than safe, since it will silently ignore malformed templates containing dangling delimiters, unmatched braces, or placeholders that are not valid Python identifiers.

\texttt{Template} instances also provide one public data attribute:

\texttt{template}

This is the object passed to the constructor’s \texttt{template} argument. In general, you shouldn’t change it, but read-only access is not enforced.

Here is an example of how to use a Template:

```python
>>> from string import Template
>>> s = Template('$$who likes $$what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $$who $$100').substitute(d)
Traceback (most recent call last):
  ...
ValueError: Invalid placeholder in string: line 1, col 11
```
Advanced usage: you can derive subclasses of Template to customize the placeholder syntax, delimiter character, or the entire regular expression used to parse template strings. To do this, you can override these class attributes:

- **delimiter** – This is the literal string describing a placeholder introducing delimiter. The default value is $. Note that this should not be a regular expression, as the implementation will call re.escape() on this string as needed.
- **idpattern** – This is the regular expression describing the pattern for non-braced placeholders (the braces will be added automatically as appropriate). The default value is the regular expression \[._a-z[._a-z0-9]*.\]

Alternatively, you can provide the entire regular expression pattern by overriding the class attribute pattern. If you do this, the value must be a regular expression object with four named capturing groups. The capturing groups correspond to the rules given above, along with the invalid placeholder rule:

- **escaped** – This group matches the escape sequence, e.g. $$, in the default pattern.
- **named** – This group matches the unbraced placeholder name; it should not include the delimiter in capturing group.
- **braced** – This group matches the brace enclosed placeholder name; it should not include either the delimiter or braces in the capturing group.
- **invalid** – This group matches any other delimiter pattern (usually a single delimiter), and it should appear last in the regular
expression.
7.1.5. String functions

The following functions are available to operate on string and Unicode objects. They are not available as string methods.

string. **capwords**(s[, sep])

Split the argument into words using `str.split()`, capitalize each word using `str.capitalize()`, and join the capitalized words using `str.join()`. If the optional second argument `sep` is absent or `None`, runs of whitespace characters are replaced by a single space and leading and trailing whitespace are removed, otherwise `sep` is used to split and join the words.

string. **maketrans**(from, to)

Return a translation table suitable for passing to `translate()`, that will map each character in `from` into the character at the same position in `to`; `from` and `to` must have the same length.

**Note:** Don’t use strings derived from `lowercase` and `uppercase` as arguments; in some locales, these don’t have the same length. For case conversions, always use `str.lower()` and `str.upper()`.
7.1.6. Deprecated string functions

The following list of functions are also defined as methods of string and Unicode objects; see section String Methods for more information on those. You should consider these functions as deprecated, although they will not be removed until Python 3. The functions defined in this module are:

```python
string.atof(s)
```

*Deprecated since version 2.0:* Use the `float()` built-in function.

Convert a string to a floating point number. The string must have the standard syntax for a floating point literal in Python, optionally preceded by a sign (± or -). Note that this behaves identical to the built-in function `float()` when passed a string.

**Note:** When passing in a string, values for NaN and Infinity may be returned, depending on the underlying C library. The specific set of strings accepted which cause these values to be returned depends entirely on the C library and is known to vary.

```python
string.atoi(s[, base])
```

*Deprecated since version 2.0:* Use the `int()` built-in function.

Convert string s to an integer in the given base. The string must consist of one or more digits, optionally preceded by a sign (± or -). The base defaults to 10. If it is 0, a default base is chosen depending on the leading characters of the string (after stripping the sign): 0x or 0X means 16, 0 means 8, anything else means 10. If base is 16, a leading 0x or 0X is always accepted, though not required. This behaves identically to the built-in function `int()` when passed a string. (Also note: for a more flexible
interpretation of numeric literals, use the built-in function `eval()`.

```python
string.atol(s[, base])
```

*Deprecated since version 2.0:* Use the `long()` built-in function.

Convert string `s` to a long integer in the given `base`. The string must consist of one or more digits, optionally preceded by a sign (+ or -). The `base` argument has the same meaning as for `atoi()`. A trailing 1 or L is not allowed, except if the base is 0. Note that when invoked without `base` or with `base` set to 10, this behaves identical to the built-in function `long()` when passed a string.

```python
string.capitalize(word)
```

Return a copy of `word` with only its first character capitalized.

```python
string.expandtabs(s[, tabsize])
```

Expand tabs in a string replacing them by one or more spaces, depending on the current column and the given tab size. The column number is reset to zero after each newline occurring in the string. This doesn’t understand other non-printing characters or escape sequences. The tab size defaults to 8.

```python
string.find(s, sub[, start[, end]])
```

Return the lowest index in `s` where the substring `sub` is found such that `sub` is wholly contained in `s[start:end]`. Return -1 on failure. Defaults for `start` and `end` and interpretation of negative values is the same as for slices.

```python
string.rfind(s, sub[, start[, end]])
```

Like `find()` but find the highest index.

```python
string.index(s, sub[, start[, end]])
```
Like `find()` but raise `ValueError` when the substring is not found.

```python
string.rindex(s, sub[, start[, end]])
```
Like `rfind()` but raise `ValueError` when the substring is not found.

```python
string.count(s, sub[, start[, end]])
```
Return the number of (non-overlapping) occurrences of substring `sub` in string `s[start:end]`. Defaults for `start` and `end` and interpretation of negative values are the same as for slices.

```python
string.lower(s)
```
Return a copy of `s`, but with upper case letters converted to lower case.

```python
string.split(s[, sep[, maxsplit]])
```
Return a list of the words of the string `s`. If the optional second argument `sep` is absent or `None`, the words are separated by arbitrary strings of whitespace characters (space, tab, newline, return, formfeed). If the second argument `sep` is present and not `None`, it specifies a string to be used as the word separator. The returned list will then have one more item than the number of non-overlapping occurrences of the separator in the string. If `maxsplit` is given, at most `maxsplit` number of splits occur, and the remainder of the string is returned as the final element of the list (thus, the list will have at most `maxsplit+1` elements). If `maxsplit` is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

The behavior of `split` on an empty string depends on the value of `sep`. If `sep` is not specified, or specified as `None`, the result will be an empty list. If `sep` is specified as any string, the result will be a list containing one element which is an empty string.
string.\textbf{rsplit}(s[, \textit{sep}[, \textit{maxsplit}]]])

Return a list of the words of the string \textit{s}, scanning \textit{s} from the end. To all intents and purposes, the resulting list of words is the same as returned by \textbf{split()}, except when the optional third argument \textit{maxsplit} is explicitly specified and nonzero. If \textit{maxsplit} is given, at most \textit{maxsplit} number of splits – the \textit{rightmost} ones – occur, and the remainder of the string is returned as the first element of the list (thus, the list will have at most \textit{maxsplit+1} elements).

\textit{New in version 2.4.}

string.\textbf{splitfields}(s[, \textit{sep}[, \textit{maxsplit}]]])

This function behaves identically to \textbf{split()}. (In the past, \textbf{split()} was only used with one argument, while \textbf{splitfields()} was only used with two arguments.)

string.\textbf{join}(\textit{words}[, \textit{sep}])

Concatenate a list or tuple of words with intervening occurrences of \textit{sep}. The default value for \textit{sep} is a single space character. It is always true that \textbf{string.join(string.split(s, sep), sep)} equals \textit{s}.

string.\textbf{joinfields}(\textit{words}[, \textit{sep}])

This function behaves identically to \textbf{join()}. (In the past, \textbf{join()} was only used with one argument, while \textbf{joinfields()} was only used with two arguments.) Note that there is no \textbf{joinfields()} method on string objects; use the \textbf{join()} method instead.

string.\textbf{lstrip}(s[, \textit{chars}])

Return a copy of the string with leading characters removed. If \textit{chars} is omitted or \texttt{None}, whitespace characters are removed. If given and not \texttt{None}, \textit{chars} must be a string; the characters in the
string will be stripped from the beginning of the string this method is called on.

*Changed in version 2.2.3:* The `chars` parameter was added. The `chars` parameter cannot be passed in earlier 2.2 versions.

**`string.rstrip(s[, chars])`**
Return a copy of the string with trailing characters removed. If `chars` is omitted or `None`, whitespace characters are removed. If given and not `None`, `chars` must be a string; the characters in the string will be stripped from the end of the string this method is called on.

*Changed in version 2.2.3:* The `chars` parameter was added. The `chars` parameter cannot be passed in earlier 2.2 versions.

**`string.strip(s[, chars])`**
Return a copy of the string with leading and trailing characters removed. If `chars` is omitted or `None`, whitespace characters are removed. If given and not `None`, `chars` must be a string; the characters in the string will be stripped from the both ends of the string this method is called on.

*Changed in version 2.2.3:* The `chars` parameter was added. The `chars` parameter cannot be passed in earlier 2.2 versions.

**`string.swapcase(s)`**
Return a copy of `s`, but with lower case letters converted to upper case and vice versa.

**`string.translate(s, table[, deletchars])`**
Delete all characters from `s` that are in `deletchars` (if present), and then translate the characters using `table`, which must be a 256-character string giving the translation for each character
value, indexed by its ordinal. If table is None, then only the character deletion step is performed.

\[
\text{string. upper(s)}
\]
Return a copy of s, but with lower case letters converted to upper case.

\[
\text{string. ljust(s, width[, fillchar])}
\]
\[
\text{string. rjust(s, width[, fillchar])}
\]
\[
\text{string. center(s, width[, fillchar])}
\]
These functions respectively left-justify, right-justify and center a string in a field of given width. They return a string that is at least \text{width} characters wide, created by padding the string s with the character \text{fillchar} (default is a space) until the given width on the right, left or both sides. The string is never truncated.

\[
\text{string. zfill(s, width)}
\]
Pad a numeric string s on the left with zero digits until the given \text{width} is reached. Strings starting with a sign are handled correctly.

\[
\text{string. replace(s, old, new[, maxreplace])}
\]
Return a copy of string s with all occurrences of substring old replaced by new. If the optional argument maxreplace is given, the first maxreplace occurrences are replaced.
This module provides regular expression matching operations similar to those found in Perl. Both patterns and strings to be searched can be Unicode strings as well as 8-bit strings.

Regular expressions use the backslash character (') to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python's usage of the same character for the same purpose in string literals; for example, to match a literal backslash, one might have to write "\\" as the pattern string, because the regular expression must be \\, and each backslash must be expressed as \\ inside a regular Python string literal.

The solution is to use Python's raw string notation for regular expression patterns; backslashes are not handled in any special way in a string literal prefixed with 'r'. So r"\n" is a two-character string containing \ and \n, while "\n" is a one-character string containing a newline. Usually patterns will be expressed in Python code using this raw string notation.

It is important to note that most regular expression operations are available as module-level functions and RegexObject methods. The functions are shortcuts that don’t require you to compile a regex object first, but miss some fine-tuning parameters.
7.2.1. Regular Expression Syntax

A regular expression (or RE) specifies a set of strings that matches it; the functions in this module let you check if a particular string matches a given regular expression (or if a given regular expression matches a particular string, which comes down to the same thing).

Regular expressions can be concatenated to form new regular expressions; if $A$ and $B$ are both regular expressions, then $AB$ is also a regular expression. In general, if a string $p$ matches $A$ and another string $q$ matches $B$, the string $pq$ will match $AB$. This holds unless $A$ or $B$ contain low precedence operations; boundary conditions between $A$ and $B$; or have numbered group references. Thus, complex expressions can easily be constructed from simpler primitive expressions like the ones described here. For details of the theory and implementation of regular expressions, consult the Friedl book referenced above, or almost any textbook about compiler construction.

A brief explanation of the format of regular expressions follows. For further information and a gentler presentation, consult the Regular Expression HOWTO.

Regular expressions can contain both special and ordinary characters. Most ordinary characters, like 'A', 'a', or '0', are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so `last` matches the string 'last'. (In the rest of this section, we'll write RE’s in this special style, usually without quotes, and strings to be matched 'in single quotes'.)

Some characters, like '|' or '(', are special. Special characters either stand for classes of ordinary characters, or affect how the
regular expressions around them are interpreted. Regular expression pattern strings may not contain null bytes, but can specify the null byte using the \number notation, e.g., '\x00'.

Repetition qualifiers (*, +, ?, {m,n}, etc) cannot be directly nested. This avoids ambiguity with the non-greedy modifier suffix ?, and with other modifiers in other implementations. To apply a second repetition to an inner repetition, parentheses may be used. For example, the expression (?:a{6})* matches any multiple of six 'a' characters.

The special characters are:

'.'

(Dot.) In the default mode, this matches any character except a newline. If the DOTALL flag has been specified, this matches any character including a newline.

'^'

(Caret.) Matches the start of the string, and in MULTILINE mode also matches immediately after each newline.

'$'

Matches the end of the string or just before the newline at the end of the string, and in MULTILINE mode also matches before a newline. foo matches both ‘foo’ and ‘foobar’, while the regular expression foo$ matches only ‘foo’. More interestingly, searching for foo.$ in 'foo1
foo2' matches ‘foo2’ normally, but ‘foo1’ in MULTILINE mode; searching for a single $ in 'foo' will find two (empty) matches: one just before the newline, and one at the end of the string.

'*'

Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. ab* will match ‘a’, ‘ab’, or ‘a’ followed by any number of ‘b’s.
'+'
Causes the resulting RE to match 1 or more repetitions of the preceding RE. \texttt{ab+} will match ‘a’ followed by any non-zero number of ‘b’s; it will not match just ‘a’.

'?
Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. \texttt{ab?} will match either ‘a’ or ‘ab’.

*?, +?, ??
The '\*', '+', and '?' qualifiers are all \textit{greedy}; they match as much text as possible. Sometimes this behaviour isn’t desired; if the RE \texttt{<.*>} is matched against \texttt{<a> b <c>}, it will match the entire string, and not just \texttt{<a>}. Adding ? after the qualifier makes it perform the match in \textit{non-greedy} or \textit{minimal} fashion; as \textit{few} characters as possible will be matched. Using the RE \texttt{<.*?>} will match only \texttt{<a>}.

\{m\}
Specifies that exactly \textit{m} copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, \texttt{a\{6\}} will match exactly six ‘a’ characters, but not five.

\{m, n\}
Causes the resulting RE to match from \textit{m} to \textit{n} repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, \texttt{a\{3, 5\}} will match from 3 to 5 ‘a’ characters. Omitting \textit{m} specifies a lower bound of zero, and omitting \textit{n} specifies an infinite upper bound. As an example, \texttt{a\{4,\}b} will match \texttt{aaaab} or a thousand ‘a’ characters followed by a ‘b’, but not \texttt{aaab}. The comma may not be omitted or the modifier would be confused with the previously described form.

\{m, n\}? Causes the resulting RE to match from \textit{m} to \textit{n} repetitions of the preceding RE, attempting to match as \textit{few} repetitions as possible. This is the non-greedy version of the previous qualifier. For
example, on the 6-character string 'aaaaaa', a{3,5} will match 5 'a' characters, while a{3,5}? will only match 3 characters.

Either escapes special characters (permitting you to match characters like '*', '?', and so forth), or signals a special sequence; special sequences are discussed below.

If you're not using a raw string to express the pattern, remember that Python also uses the backslash as an escape sequence in string literals; if the escape sequence isn't recognized by Python's parser, the backslash and subsequent character are included in the resulting string. However, if Python would recognize the resulting sequence, the backslash should be repeated twice. This is complicated and hard to understand, so it's highly recommended that you use raw strings for all but the simplest expressions.

[]

Used to indicate a set of characters. In a set:

- Characters can be listed individually, e.g. [amk] will match 'a', 'm', or 'k'.
- Ranges of characters can be indicated by giving two characters and separating them by a '-', for example [a-z] will match any lowercase ASCII letter, [0-5][0-9] will match all the two-digits numbers from 00 to 59, and [0-9A-Fa-f] will match any hexadecimal digit. If - is escaped (e.g. [a\-z]) or if it's placed as the first or last character (e.g. [a-]), it will match a literal '-'.
- Special characters lose their special meaning inside sets. For example, [((+))] will match any of the literal characters '(', '+', '+', or ')'.
- Character classes such as \w or \s (defined below) are also
accepted inside a set, although the characters they match depends on whether LOCALE or UNICODE mode is in force.

- Characters that are not within a range can be matched by complementing the set. If the first character of the set is '^', all the characters that are not in the set will be matched. For example, [^5] will match any character except '5', and [^[^] will match any character except '^'. ^ has no special meaning if it's not the first character in the set.
- To match a literal ']' inside a set, precede it with a backslash, or place it at the beginning of the set. For example, both [(\[\]){}] and [[][{}]] will both match a parenthesis.

A|B, where A and B can be arbitrary REs, creates a regular expression that will match either A or B. An arbitrary number of REs can be separated by the '|' in this way. This can be used inside groups (see below) as well. As the target string is scanned, REs separated by '|' are tried from left to right. When one pattern completely matches, that branch is accepted. This means that once A matches, B will not be tested further, even if it would produce a longer overall match. In other words, the '|' operator is never greedy. To match a literal '|', use \|, or enclose it inside a character class, as in [\|].

(...)
Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the \number special sequence, described below. To match the literals '( ' or ' )', use \( or \), or enclose them inside a character class: [([ ])].

(?...) This is an extension notation (a '?' following a '(' is not
meaningful otherwise). The first character after the '?' determines what the meaning and further syntax of the construct is. Extensions usually do not create a new group; (?P<name>...) is the only exception to this rule. Following are the currently supported extensions.

(?iLmsux)

(One or more letters from the set 'i', 'L', 'm', 's', 'u', 'x'.) The group matches the empty string; the letters set the corresponding flags: re.I (ignore case), re.L (locale dependent), re.M (multi-line), re.S (dot matches all), re.U (Unicode dependent), and re.X (verbose), for the entire regular expression. (The flags are described in Module Contents.) This is useful if you wish to include the flags as part of the regular expression, instead of passing a flag argument to the re.compile() function.

Note that the (?x) flag changes how the expression is parsed. It should be used first in the expression string, or after one or more whitespace characters. If there are non-whitespace characters before the flag, the results are undefined.

(?:...) A non-capturing version of regular parentheses. Matches whatever regular expression is inside the parentheses, but the substring matched by the group cannot be retrieved after performing a match or referenced later in the pattern.

(?P<name>...) Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name name. Group names must be valid Python identifiers, and each group name must be defined only once within a regular expression. A symbolic group is also a numbered group, just as if the group were not named.
Named groups can be referenced in three contexts. If the pattern is `(\P<quote>['"]\*)\*(\P=quote)` (i.e. matching a string quoted with either single or double quotes):

<table>
<thead>
<tr>
<th>Context of reference to group “quote”</th>
<th>Ways to reference it</th>
</tr>
</thead>
</table>
| in the same pattern itself            | • (?P=quote) (as shown)  
• \1                                      |
| when processing match object m        | • m.group('quote')  
• m.end('quote') (etc.) |
| in a string passed to the repl argument of re.sub() | • \g<quote>  
• \g<1>  
• \1 |

(?P=name)
A backreference to a named group; it matches whatever text was matched by the earlier group named `name`.

(?#...)
A comment; the contents of the parentheses are simply ignored.

(?=...)
Matches if `...` matches next, but doesn’t consume any of the string. This is called a lookahead assertion. For example, `Isaac (?=Asimov)` will match 'Isaac ' only if it’s followed by 'Asimov'.

(?!...)
Matches if `...` doesn’t match next. This is a negative lookahead assertion. For example, `Isaac (?!Asimov)` will match 'Isaac ' only if it’s not followed by 'Asimov'.

(?<=...)
Matches if the current position in the string is preceded by a match for `...` that ends at the current position. This is called a *positive lookbehind assertion*. `(=?=abc)def` will find a match in
abcdef, since the lookbehind will back up 3 characters and check if the contained pattern matches. The contained pattern must only match strings of some fixed length, meaning that abc or a|b are allowed, but a* and a{3,4} are not. Group references are not supported even if they match strings of some fixed length. Note that patterns which start with positive lookbehind assertions will not match at the beginning of the string being searched; you will most likely want to use the search() function rather than the match() function:

```python
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

This example looks for a word following a hyphen:

```python
>>> m = re.search('(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

(?<!...) Matches if the current position in the string is not preceded by a match for .... This is called a *negative lookbehind assertion*. Similar to positive lookbehind assertions, the contained pattern must only match strings of some fixed length and shouldn’t contain group references. Patterns which start with negative lookbehind assertions may match at the beginning of the string being searched.

(?id/name)yes-pattern|no-pattern Will try to match with yes-pattern if the group with given id or name exists, and with no-pattern if it doesn’t. no-pattern is optional and can be omitted. For example, (<)?(\w*@\w+(?:\.\w+)+)?(1)> is a poor email matching pattern, which will match with '<user@host.com>' as well as 'user@host.com', but
not with \'<user@host.com'.

*New in version 2.4.*

The special sequences consist of `\` and a character from the list below. If the ordinary character is not on the list, then the resulting RE will match the second character. For example, `\$` matches the character `'$'`.

`\number`

Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, `(.+)\1` matches `the the` or `55 55`, but not `the the` (note the space after the group). This special sequence can only be used to match one of the first 99 groups. If the first digit of `number` is 0, or `number` is 3 octal digits long, it will not be interpreted as a group match, but as the character with octal value `number`. Inside the `['` and `']` of a character class, all numeric escapes are treated as characters.

`\A`

Matches only at the start of the string.

`\b`

Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of alphanumeric or underscore characters, so the end of a word is indicated by whitespace or a non-alphanumeric, non-underscore character. Note that formally, `\b` is defined as the boundary between a `\w` and a `\w` character (or vice versa), or between `\w` and the beginning/end of the string, so the precise set of characters deemed to be alphanumeric depends on the values of the `UNICODE` and `LOCALE` flags. For example, `r'\bfoo\b'` matches `foo`, `foo.` , `'(foo)'`, `bar foo baz` but not `foobar` or `foo3`. Inside a character range, `\b` represents the backspace character, for compatibility with Python’s string literals.
\B
Matches the empty string, but only when it is not at the beginning or end of a word. This means that `r'py\B'` matches `python`, `py3`, `py2`, but not `py`, `py.` , or `py!'`. \B is just the opposite of \b, so is also subject to the settings of \locale and unicode.

\d
When the \texttt{UNICODE} flag is not specified, matches any decimal digit; this is equivalent to the set \texttt{[0-9]}. With \texttt{UNICODE}, it will match whatever is classified as a decimal digit in the Unicode character properties database.

\D
When the \texttt{UNICODE} flag is not specified, matches any non-digit character; this is equivalent to the set \texttt{[^0-9]}. With \texttt{UNICODE}, it will match anything other than character marked as digits in the Unicode character properties database.

\s
When the \texttt{UNICODE} flag is not specified, it matches any whitespace character, this is equivalent to the set \texttt{[\t\n\r\f\v]}. The \texttt{LOCALE} flag has no extra effect on matching of the space. If \texttt{UNICODE} is set, this will match the characters \texttt{[\t\n\r\f\v]} plus whatever is classified as space in the Unicode character properties database.

\S
When the \texttt{UNICODE} flag is not specified, matches any non-whitespace character; this is equivalent to the set \texttt{[^\t\n\r\f\v]}. The \texttt{LOCALE} flag has no extra effect on non-whitespace match. If \texttt{UNICODE} is set, then any character not marked as space in the Unicode character properties database is matched.

\w
When the \texttt{LOCALE} and \texttt{UNICODE} flags are not specified, matches any alphanumeric character and the underscore; this is
equivalent to the set \([a-zA-Z0-9_]\). With \texttt{locale}, it will match the set \([0-9_]\) plus whatever characters are defined as alphanumeric for the current locale. If \texttt{unicode} is set, this will match the characters \([0-9_]\) plus whatever is classified as alphanumeric in the Unicode character properties database.

\texttt{\W}

When the \texttt{locale} and \texttt{unicode} flags are not specified, matches any non-alphanumeric character; this is equivalent to the set \(^{\text{^a-zA-Z0-9}}\). With \texttt{locale}, it will match any character not in the set \([0-9_]\), and not defined as alphanumeric for the current locale. If \texttt{unicode} is set, this will match anything other than \([0-9_]\) plus characters classified as not alphanumeric in the Unicode character properties database.

\texttt{\Z}

Matches only at the end of the string.

If both \texttt{locale} and \texttt{unicode} flags are included for a particular sequence, then \texttt{locale} flag takes effect first followed by the \texttt{unicode}.

Most of the standard escapes supported by Python string literals are also accepted by the regular expression parser:

\begin{verbatim}
\a  \b  \f  \n
\r  \t  \v  \x
\end{verbatim}

(Note that \texttt{\b} is used to represent word boundaries, and means “backspace” only inside character classes.)

Octal escapes are included in a limited form: If the first digit is a 0, or if there are three octal digits, it is considered an octal escape. Otherwise, it is a group reference. As for string literals, octal escapes are always at most three digits in length.
See also:

Mastering Regular Expressions
   Book on regular expressions by Jeffrey Friedl, published by O'Reilly. The second edition of the book no longer covers Python at all, but the first edition covered writing good regular expression patterns in great detail.
The module defines several functions, constants, and an exception. Some of the functions are simplified versions of the full featured methods for compiled regular expressions. Most non-trivial applications always use the compiled form.

\texttt{re.compile(pattern, flags=0)}

Compile a regular expression pattern into a regular expression object, which can be used for matching using its \texttt{match()} and \texttt{search()} methods, described below.

The expression’s behaviour can be modified by specifying a \texttt{flags} value. Values can be any of the following variables, combined using bitwise OR (the | operator).

The sequence

\begin{verbatim}
prog = re.compile(pattern)
result = prog.match(string)
\end{verbatim}

is equivalent to

\begin{verbatim}
result = re.match(pattern, string)
\end{verbatim}

but using \texttt{re.compile()} and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

\textbf{Note:} The compiled versions of the most recent patterns passed to \texttt{re.match()}, \texttt{re.search()} or \texttt{re.compile()} are cached, so programs that use only a few regular expressions at a time needn’t worry about compiling regular expressions.
re. **DEBUG**

Display debug information about compiled expression.

re. **I**
re. **IGNORECASE**

Perform case-insensitive matching; expressions like `\[A-Z\]` will match lowercase letters, too. This is not affected by the current locale.

re. **L**
re. **LOCALE**

Make `\w`, `\W`, `\b`, `\B`, `\s` and `\S` dependent on the current locale.

re. **M**
re. **MULTILINE**

When specified, the pattern character `'^'` matches at the beginning of the string and at the beginning of each line (immediately following each newline); and the pattern character `'$'` matches at the end of the string and at the end of each line (immediately preceding each newline). By default, `'^'` matches only at the beginning of the string, and `'$'` only at the end of the string and immediately before the newline (if any) at the end of the string.

re. **S**
re. **DOTALL**

Make the `.' special character match any character at all, including a newline; without this flag, `.' will match anything except a newline.

re. **U**
re. **UNICODE**

Make `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` and `\S` dependent on the Unicode character properties database.

*New in version 2.0.*
re.X
re.VERBOSE

This flag allows you to write regular expressions that look nicer and are more readable by allowing you to visually separate logical sections of the pattern and add comments. Whitespace within the pattern is ignored, except when in a character class or when preceded by an unescaped backslash. When a line contains a # that is not in a character class and is not preceded by an unescaped backslash, all characters from the leftmost such # through the end of the line are ignored.

This means that the two following regular expression objects that match a decimal number are functionally equal:

```python
a = re.compile(r"""\d +  # the integral part
\ .  # the decimal point
\d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

re.search(pattern, string, flags=0)

Scan through string looking for the first location where the regular expression pattern produces a match, and return a corresponding MatchObject instance. Return None if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

re.match(pattern, string, flags=0)

If zero or more characters at the beginning of string match the regular expression pattern, return a corresponding MatchObject instance. Return None if the string does not match the pattern; note that this is different from a zero-length match.

Note that even in MULTILINE mode, re.match() will only match at the beginning of the string and not at the beginning of each line.

If you want to locate a match anywhere in string, use search()
instead (see also search() vs. match()

re. split(pattern, string, maxsplit=0, flags=0)

Split string by the occurrences of pattern. If capturing parentheses are used in pattern, then the text of all groups in the pattern are also returned as part of the resulting list. If maxsplit is nonzero, at most maxsplit splits occur, and the remainder of the string is returned as the final element of the list. (Incompatibility note: in the original Python 1.5 release, maxsplit was ignored. This has been fixed in later releases.)

```python
>>> re.split('\W+', 'Words, words, words."
['Words', 'words', 'words', '"

>>> re.split('(\W+)\', 'Words, words, words."
['Words', ' ', 'words', ' ', 'words', ' .', '"

>>> re.split('\W+', 'Words, words, words.', 1)
['Words', 'words, words."

>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

If there are capturing groups in the separator and it matches at the start of the string, the result will start with an empty string. The same holds for the end of the string:

```python
>>> re.split('(\W+)\', '...words, words...
[' ', ' ...', ' words', ' ', ' words', ' ...
', '"

That way, separator components are always found at the same relative indices within the result list (e.g., if there's one capturing group in the separator, the 0th, the 2nd and so forth).

Note that split will never split a string on an empty pattern match. For example:

```python
>>> re.split('x*', 'foo')
['foo']

>>> re.split("(?m)^$", "foo\n\nbar\n")
['foo\n\nbar\n']
**Changed in version 2.7:** Added the optional flags argument.

```python
def findall(pattern, string, flags=0)
```

Return all non-overlapping matches of *pattern* in *string*, as a list of strings. The *string* is scanned left-to-right, and matches are returned in the order found. If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group. Empty matches are included in the result unless they touch the beginning of another match.

*New in version 1.5.2.*

**Changed in version 2.4:** Added the optional flags argument.

```python
def finditer(pattern, string, flags=0)
```

Return an iterator yielding *MatchObject* instances over all non-overlapping matches for the RE *pattern* in *string*. The *string* is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result unless they touch the beginning of another match.

*New in version 2.2.*

**Changed in version 2.4:** Added the optional flags argument.

```python
def sub(pattern, repl, string, count=0, flags=0)
```

Return the string obtained by replacing the leftmost non-overlapping occurrences of *pattern* in *string* by the replacement *repl*. If the pattern isn’t found, *string* is returned unchanged. *repl* can be a string or a function; if it is a string, any backslash escapes in it are processed. That is, \n is converted to a single newline character, \r is converted to a carriage return, and so forth. Unknown escapes such as \j are left alone. Backreferences, such as \6, are replaced with the substring matched by group 6 in the pattern. For example:
>>> re.sub(re'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s+([^\s])\:', 
...     r'static PyObject*\npy_\1(void)\n{', 
...     'def myfunc():', 
'static PyObject*\npy_myfunc(void)\n{'

If \textit{repl} is a function, it is called for every non-overlapping occurrence of \textit{pattern}. The function takes a single match object argument, and returns the replacement string. For example:

```python
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-':
...         return ''
...     else:
...         return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro---gram-files')
'pro--gram files'
>>> re.sub('sANDs', ' & ', 'Baked Beans And Spam', flags=re)  
'Baked Beans & Spam'
```

The pattern may be a string or an RE object.

The optional argument \textit{count} is the maximum number of pattern occurrences to be replaced; \textit{count} must be a non-negative integer. If omitted or zero, all occurrences will be replaced. Empty matches for the pattern are replaced only when not adjacent to a previous match, so \texttt{re.sub('x*',' - ','abc')} returns \texttt{'-a-b-c-'}.

In string-type \texttt{repl} arguments, in addition to the character escapes and backreferences described above, \texttt{	extbackslash g<name>} will use the substring matched by the group named \textit{name}, as defined by the \texttt{(\?P<name>...)} syntax. \texttt{\g<number>} uses the corresponding group number; \texttt{\g<2>} is therefore equivalent to \texttt{\2}, but isn’t ambiguous in a replacement such as \texttt{\g<2>0}. \texttt{\20} would be interpreted as a reference to group 20, not a reference to group 2 followed by the literal character '0'. The backreference \texttt{\g<0>} substitutes in the entire substring matched by the RE.

\textit{Changed in version 2.7}: Added the optional flags argument.
re. **subn**(*pattern*, *repl*, *string*, *count*=0, *flags*=0)

Perform the same operation as sub(), but return a tuple (new_string, number_of_subs_made).

*Changed in version 2.7:* Added the optional flags argument.

re. **escape**(*string*)

Return string with all non-alphanumerics backslashed; this is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it.

re. **purge**()

Clear the regular expression cache.

**exception** re. **error**

Exception raised when a string passed to one of the functions here is not a valid regular expression (for example, it might contain unmatched parentheses) or when some other error occurs during compilation or matching. It is never an error if a string contains no match for a pattern.
7.2.3. Regular Expression Objects

class `re.RegexObject`

The `RegexObject` class supports the following methods and attributes:

`search(string[, pos[, endpos]])`

Scan through `string` looking for a location where this regular expression produces a match, and return a corresponding `MatchObject` instance. Return `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

The optional second parameter `pos` gives an index in the string where the search is to start; it defaults to 0. This is not completely equivalent to slicing the string; the `'^'` pattern character matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start.

The optional parameter `endpos` limits how far the string will be searched; it will be as if the string is `endpos` characters long, so only the characters from `pos` to `endpos - 1` will be searched for a match. If `endpos` is less than `pos`, no match will be found, otherwise, if `rx` is a compiled regular expression object, `rx.search(string, 0, 50)` is equivalent to `rx.search(string[:50], 0).

```python
g>>> pattern = re.compile("d")
g>>> pattern.search("dog")  # Match at index 0
<_sre.SRE_Match object at ...>
g>>> pattern.search("dog", 1)  # No match; search doesn't
```
match(string[, pos[, endpos]])
If zero or more characters at the beginning of string match this
regular expression, return a corresponding MatchObject
instance. Return None if the string does not match the pattern;
note that this is different from a zero-length match.

The optional pos and endpos parameters have the same
meaning as for the search() method.

```python
>>> pattern = re.compile("o")
>>> pattern.match("dog")  # No match as "o" is not at the start
>>> pattern.match("dog", 1)  # Match as "o" is the 2nd character
<_sre.SRE_Match object at ...>
```

If you want to locate a match anywhere in string, use search() instead (see also search() vs. match()).

split(string, maxsplit=0)
Identical to the split() function, using the compiled pattern.

findall(string[, pos[, endpos]])
Similar to the findall() function, using the compiled pattern,
but also accepts optional pos and endpos parameters that
limit the search region like for match().

finditer(string[, pos[, endpos]])
Similar to the finditer() function, using the compiled pattern,
but also accepts optional pos and endpos parameters that
limit the search region like for match().

sub(repl, string, count=0)
Identical to the sub() function, using the compiled pattern.

subn(repl, string, count=0)
Identical to the `subn()` function, using the compiled pattern.

**flags**
The regex matching flags. This is a combination of the flags given to `compile()` and any (?...) inline flags in the pattern.

**groups**
The number of capturing groups in the pattern.

**groupindex**
A dictionary mapping any symbolic group names defined by (?P<id>) to group numbers. The dictionary is empty if no symbolic groups were used in the pattern.

**pattern**
The pattern string from which the RE object was compiled.
7.2.4. Match Objects

class re.MatchObject

Match objects always have a boolean value of True. Since match() and search() return None when there is no match, you can test whether there was a match with a simple if statement:

```python
match = re.search(pattern, string)
if match:
    process(match)
```

Match objects support the following methods and attributes:

**expand(template)**

Return the string obtained by doing backslash substitution on the template string template, as done by the sub() method. Escapes such as \n are converted to the appropriate characters, and numeric backreferences (\1, \2) and named backreferences (\g<1>, \g<name>) are replaced by the contents of the corresponding group.

**group([group1, ...])**

Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, group1 defaults to zero (the whole match is returned). If a groupN argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized group. If a group number is negative or larger than the number of groups defined in the pattern, an IndexError exception is raised. If a group is contained in a part of the pattern that did not match, the
corresponding result is \texttt{None}. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.

```python
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)  # The entire match
'Isaac Newton'
>>> m.group(1)  # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)  # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)  # Multiple arguments give us a tuple
('Isaac', 'Newton')
```

If the regular expression uses the \texttt{(?P<name>...)} syntax, the \texttt{groupN} arguments may also be strings identifying groups by their group name. If a string argument is not used as a group name in the pattern, an \texttt{IndexError} exception is raised.

A moderately complicated example:

```python
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

Named groups can also be referred to by their index:

```python
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

If a group matches multiple times, only the last match is accessible:

```python
>>> m = re.match(r"(\.)++", "a1b2c3")  # Matches 3 times.
>>> m.group(1)  # Returns only the
'M
a1
b2
c3'
```
groups([default])
Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The default argument is used for groups that did not participate in the match; it defaults to None. (Incompatibility note: in the original Python 1.5 release, if the tuple was one element long, a string would be returned instead. In later versions (from 1.5.1 on), a singleton tuple is returned in such cases.)

For example:

```python
>>> m = re.match(r"(?P<first_name>\w+) \(?P<last_name>\w+\)", "Malcolm Reynolds")
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

If we make the decimal place and everything after it optional, not all groups might participate in the match. These groups will default to None unless the default argument is given:

```python
>>> m = re.match(r"(?P<first_name>\w+) \?(\?\w+)?", "Malcolm Reynolds")
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

groupdict([default])
Return a dictionary containing all the named subgroups of the match, keyed by the subgroup name. The default argument is used for groups that did not participate in the match; it defaults to None. For example:

```python
>>> m = re.match(r"(?P<first_name>\w+) \(?P<last_name>\w+)", "Malcolm Reynolds")
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```
\texttt{start([group])}

\texttt{end([group])}

Return the indices of the start and end of the substring matched by \texttt{group}; \texttt{group} defaults to zero (meaning the whole matched substring). Return \texttt{-1} if \texttt{group} exists but did not contribute to the match. For a match object \texttt{m}, and a group \texttt{g} that did contribute to the match, the substring matched by group \texttt{g} (equivalent to \texttt{m.group(g)}) is

\texttt{m.string[m.start(g):m.end(g)]}

Note that \texttt{m.start(group)} will equal \texttt{m.end(group)} if \texttt{group} matched a null string. For example, after \texttt{m = re.search('b(c?)', 'cba')}, \texttt{m.start(0)} is 1, \texttt{m.end(0)} is 2, \texttt{m.start(1)} and \texttt{m.end(1)} are both 2, and \texttt{m.start(2)} raises an \texttt{IndexError} exception.

An example that will remove \texttt{remove_this} from email addresses:

```python
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

\texttt{span([group])}

For \texttt{MatchObject m}, return the 2-tuple \texttt{(m.start(group), m.end(group))}. Note that if \texttt{group} did not contribute to the match, this is \texttt{(-1, -1)}. \texttt{group} defaults to zero, the entire match.

\texttt{pos}

The value of \texttt{pos} which was passed to the \texttt{search()} or
match() method of the RegexObject. This is the index into the string at which the RE engine started looking for a match.

endpos
The value of endpos which was passed to the search() or match() method of the RegexObject. This is the index into the string beyond which the RE engine will not go.

lastindex
The integer index of the last matched capturing group, or None if no group was matched at all. For example, the expressions (a)b, ((a)(b)), and ((ab)) will have lastindex == 1 if applied to the string 'ab', while the expression (a)(b) will have lastindex == 2, if applied to the same string.

lastgroup
The name of the last matched capturing group, or None if the group didn't have a name, or if no group was matched at all.

re
The regular expression object whose match() or search() method produced this MatchObject instance.

string
The string passed to match() or search().
7.2.5. Examples

7.2.5.1. Checking For a Pair

In this example, we'll use the following helper function to display match objects a little more gracefully:

```python
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

Suppose you are writing a poker program where a player's hand is represented as a 5-character string with each character representing a card, “a” for ace, “k” for king, “q” for queen, “j” for jack, “t” for 10, and “2” through “9” representing the card with that value.

To see if a given string is a valid hand, one could do the following:

```python
>>> valid = re.compile(r'^[a2-9tjqk]{5}$')
>>> displaymatch(valid.match('akt5q'))  # Valid.
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match('akt5e'))  # Invalid.
>>> displaymatch(valid.match('akt'))  # Invalid.
>>> displaymatch(valid.match('727ak'))  # Valid.
"<Match: '727ak', groups=()>"
```

That last hand, "727ak", contained a pair, or two of the same valued cards. To match this with a regular expression, one could use backreferences as such:

```python
>>> pair = re.compile(r".*(.).*\1")
>>> displaymatch(pair.match('717ak'))  # Pair of 7s.
"<Match: '717', groups=('7',)>"
>>> displaymatch(pair.match('718ak'))  # No pairs.
>>> displaymatch(pair.match('354aa'))  # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```
To find out what card the pair consists of, one could use the **group()** method of `MatchObject` in the following manner:

```python
>>> pair.match("717ak").group(1)
'7'
# Error because re.match() returns None, which doesn't have a group()
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.).*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'
```

7.2.5.2. Simulating `scanf()`

Python does not currently have an equivalent to `scanf()`. Regular expressions are generally more powerful, though also more verbose, than `scanf()` format strings. The table below offers some more-or-less equivalent mappings between `scanf()` format tokens and regular expressions.

<table>
<thead>
<tr>
<th><code>scanf()</code> Token</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>.</td>
</tr>
<tr>
<td>%5c</td>
<td>.{5}</td>
</tr>
<tr>
<td>%d</td>
<td>+[-]?d+</td>
</tr>
</tbody>
</table>
| %e, %E, %f, %g | \+[\-]?([d+(\.[d]*)?|\.[d+]([eE]\+[\-]?[d+]?)?
| %i              | \+[\-]?([0][xX][dA-Fa-f]+|0[0-7]*|d+) |
| %o              | \+[\-]?[0-7]+     |
| %s              | \S+                |
| %u              | \d+                |
| %x, %X         | \+[\-]?([0][xX])?[dA-Fa-f]+ |
To extract the filename and numbers from a string like

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

you would use a `scanf()` format like

```
%*s - %d errors, %d warnings
```

The equivalent regular expression would be

```
(\S+) - (\d+) errors, (\d+) warnings
```

### 7.2.5.3. search() vs. match()

Python offers two different primitive operations based on regular expressions: `re.match()` checks for a match only at the beginning of the string, while `re.search()` checks for a match anywhere in the string (this is what Perl does by default).

For example:

```python
>>> re.match("c", "abcdef")    # No match
>>> re.search("c", "abcdef")  # Match
<_sre.SRE_Match object at ...>  
```

Regular expressions beginning with `\^` can be used with `search()` to restrict the match at the beginning of the string:

```python
>>> re.match("c", "abcdef")    # No match
>>> re.search("\^c", "abcdef") # No match
>>> re.search("\^a", "abcdef") # Match
<_sre.SRE_Match object at ...>
```

Note however that in `MULTILINE` mode `match()` only matches at the beginning of the string, whereas using `search()` with a regular expression beginning with `\^` will match at the beginning of each
7.2.5.4. Making a Phonebook

`split()` splits a string into a list delimited by the passed pattern. The method is invaluable for converting textual data into data structures that can be easily read and modified by Python as demonstrated in the following example that creates a phonebook.

First, here is the input. Normally it may come from a file, here we are using triple-quoted string syntax:

```python
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
... Heather Albrecht: 548.326.4584 919 Park Place"
```

The entries are separated by one or more newlines. Now we convert the string into a list with each nonempty line having its own entry:

```python
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
 'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
 'Frank Burger: 925.541.7625 662 South Dogwood Way',
 'Heather Albrecht: 548.326.4584 919 Park Place']
```

Finally, split each entry into a list with first name, last name, telephone number, and address. We use the `maxsplit` parameter of `split()` because the address has spaces, our splitting pattern, in it:
The `:`? pattern matches the colon after the last name, so that it does not occur in the result list. With a maxsplit of 4, we could separate the house number from the street name:

```python
>>> [re.split("?: ", entry, 3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

### 7.2.5.5. Text Munging

`sub()` replaces every occurrence of a pattern with a string or the result of a function. This example demonstrates using `sub()` with a function to “munge” text, or randomize the order of all the characters in each word of a sentence except for the first and last characters:

```python
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + ''.join(inner_word) + m.group(3)

>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + ''.join(inner_word) + m.group(3)

>>> text = "Professor Abdolmalek, please report your absences promptly."

>>> re.sub(r"\w+(\w+)?\w+", repl, text)
'Poefsroser Aodlambelk, pslaee rerorpt yuor abnsebes plmrptoy.'

>>> re.sub(r"\w+(\w+)?\w+", repl, text)
'Pofsroser Aodlambelk, plaese reoprpt yuor asnebces potlmrpy.'
```

### 7.2.5.6. Finding all Adverbs

`findall()` matches all occurrences of a pattern, not just the first one.
as `search()` does. For example, if one was a writer and wanted to find all of the adverbs in some text, he or she might use `findall()` in the following manner:

```python
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly", text)
['carefully', 'quickly']
```

### 7.2.5.7. Finding all Adverbs and their Positions

If one wants more information about all matches of a pattern than the matched text, `finditer()` is useful as it provides instances of `MatchObject` instead of strings. Continuing with the previous example, if one was a writer who wanted to find all of the adverbs and their positions in some text, he or she would use `finditer()` in the following manner:

```python
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly", text):
...     print '%02d-%02d: %s % (m.start(), m.end(), m.group(0))
07-16: carefully
40-47: quickly
```

### 7.2.5.8. Raw String Notation

Raw string notation (`r"text"`) keeps regular expressions sane. Without it, every backslash (`'\'`) in a regular expression would have to be prefixed with another one to escape it. For example, the two following lines of code are functionally identical:

```python
>>> re.match(r"\W(.)\1\W", " ff ")
<_sre.SRE_Match object at ...>  
>>> re.match("\\W(.)\\1\\W", " ff ")
<_sre.SRE_Match object at ...>
```
When one wants to match a literal backslash, it must be escaped in the regular expression. With raw string notation, this means `r"\\"`. Without raw string notation, one must use `"\\\"`, making the following lines of code functionally identical:

```python
>>> re.match(r"\\\", r"\\")
<_sre.SRE_Match object at ...>
>>> re.match("\\\\", r"\\")
<_sre.SRE_Match object at ...>
```
7.3. **struct** — Interpret strings as packed binary data

This module performs conversions between Python values and C structs represented as Python strings. This can be used in handling binary data stored in files or from network connections, among other sources. It uses **Format Strings** as compact descriptions of the layout of the C structs and the intended conversion to/from Python values.

**Note:** By default, the result of packing a given C struct includes pad bytes in order to maintain proper alignment for the C types involved; similarly, alignment is taken into account when unpacking. This behavior is chosen so that the bytes of a packed struct correspond exactly to the layout in memory of the corresponding C struct. To handle platform-independent data formats or omit implicit pad bytes, use *standard* size and alignment instead of *native* size and alignment: see **Byte Order, Size, and Alignment** for details.
7.3.1. Functions and Exceptions

The module defines the following exception and functions:

**exception** `struct.error`

Exception raised on various occasions; argument is a string describing what is wrong.

`struct.pack(fmt, v1, v2, ...)`

Return a string containing the values `v1`, `v2`, ... packed according to the given format. The arguments must match the values required by the format exactly.

`struct.pack_into(fmt, buffer, offset, v1, v2, ...)`

Pack the values `v1`, `v2`, ... according to the given format, write the packed bytes into the writable `buffer` starting at `offset`. Note that the offset is a required argument.

*New in version 2.5.*

`struct.unpack(fmt, string)`

Unpack the string (presumably packed by `pack(fmt, ...)`) according to the given format. The result is a tuple even if it contains exactly one item. The string must contain exactly the amount of data required by the format (`len(string)` must equal `calcsize(fmt)`).

`struct.unpack_from(fmt, buffer[, offset=0])`

Unpack the `buffer` according to the given format. The result is a tuple even if it contains exactly one item. The `buffer` must contain at least the amount of data required by the format (`len(buffer[offset:])`) must be at least `calcsize(fmt)`.)
New in version 2.5.

```
struct calcsize(fmt)
    Return the size of the struct (and hence of the string) corresponding to the given format.
```
7.3.2. Format Strings

Format strings are the mechanism used to specify the expected layout when packing and unpacking data. They are built up from **Format Characters**, which specify the type of data being packed/unpacked. In addition, there are special characters for controlling the **Byte Order, Size, and Alignment**.

7.3.2.1. Byte Order, Size, and Alignment

By default, C types are represented in the machine's native format and byte order, and properly aligned by skipping pad bytes if necessary (according to the rules used by the C compiler).

Alternatively, the first character of the format string can be used to indicate the byte order, size and alignment of the packed data, according to the following table:

<table>
<thead>
<tr>
<th>Character</th>
<th>Byte order</th>
<th>Size</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>native</td>
<td>native</td>
<td>native</td>
</tr>
<tr>
<td>=</td>
<td>native</td>
<td>standard</td>
<td>none</td>
</tr>
<tr>
<td>&lt;</td>
<td>little-endian</td>
<td>standard</td>
<td>none</td>
</tr>
<tr>
<td>&gt;</td>
<td>big-endian</td>
<td>standard</td>
<td>none</td>
</tr>
<tr>
<td>!</td>
<td>network (= big-endian)</td>
<td>standard</td>
<td>none</td>
</tr>
</tbody>
</table>

If the first character is not one of these, `@` is assumed.

Native byte order is big-endian or little-endian, depending on the host system. For example, Intel x86 and AMD64 (x86-64) are little-endian; Motorola 68000 and PowerPC G5 are big-endian; ARM and Intel Itanium feature switchable endianness (bi-endian). Use `sys.byteorder` to check the endianness of your system.
Native size and alignment are determined using the C compiler’s `sizeof` expression. This is always combined with native byte order.

Standard size depends only on the format character; see the table in the Format Characters section.

Note the difference between '@' and '=': both use native byte order, but the size and alignment of the latter is standardized.

The form '!' is available for those poor souls who claim they can’t remember whether network byte order is big-endian or little-endian.

There is no way to indicate non-native byte order (force byte-swapping); use the appropriate choice of '<' or '>'.

Notes:

1. Padding is only automatically added between successive structure members. No padding is added at the beginning or the end of the encoded struct.
2. No padding is added when using non-native size and alignment, e.g. with '<', '>', '=', and '!'.
3. To align the end of a structure to the alignment requirement of a particular type, end the format with the code for that type with a repeat count of zero. See Examples.

### 7.3.2.2. Format Characters

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types. The ‘Standard size’ column refers to the size of the packed value in bytes when using standard size; that is, when the format string starts with one of '<', '>', '!' or '='. When using native size, the size of the packed value is platform-dependent.
<table>
<thead>
<tr>
<th>Format</th>
<th>C Type</th>
<th>Python type</th>
<th>Standard size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>pad byte</td>
<td>no value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>char</td>
<td>string of length 1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>signed char</td>
<td>integer</td>
<td>1</td>
<td>(3)</td>
</tr>
<tr>
<td>B</td>
<td>unsigned char</td>
<td>integer</td>
<td>1</td>
<td>(3)</td>
</tr>
<tr>
<td>?</td>
<td>_Bool</td>
<td>bool</td>
<td>1</td>
<td>(1)</td>
</tr>
<tr>
<td>h</td>
<td>short</td>
<td>integer</td>
<td>2</td>
<td>(3)</td>
</tr>
<tr>
<td>H</td>
<td>unsigned short</td>
<td>integer</td>
<td>2</td>
<td>(3)</td>
</tr>
<tr>
<td>i</td>
<td>int</td>
<td>integer</td>
<td>4</td>
<td>(3)</td>
</tr>
<tr>
<td>I</td>
<td>unsigned int</td>
<td>integer</td>
<td>4</td>
<td>(3)</td>
</tr>
<tr>
<td>l</td>
<td>long</td>
<td>integer</td>
<td>4</td>
<td>(3)</td>
</tr>
<tr>
<td>L</td>
<td>unsigned long</td>
<td>integer</td>
<td>4</td>
<td>(3)</td>
</tr>
<tr>
<td>q</td>
<td>long long</td>
<td>integer</td>
<td>8</td>
<td>(2), (3)</td>
</tr>
<tr>
<td>Q</td>
<td>unsigned long long</td>
<td>integer</td>
<td>8</td>
<td>(2), (3)</td>
</tr>
<tr>
<td>f</td>
<td>float</td>
<td>float</td>
<td>4</td>
<td>(4)</td>
</tr>
<tr>
<td>d</td>
<td>double</td>
<td>float</td>
<td>8</td>
<td>(4)</td>
</tr>
<tr>
<td>s</td>
<td>char[]</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>char[]</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>void *</td>
<td>integer</td>
<td></td>
<td>(5), (3)</td>
</tr>
</tbody>
</table>

Notes:

1. The '?\' conversion code corresponds to the _Bool type defined by C99. If this type is not available, it is simulated using a char. In standard mode, it is always represented by one byte.

   *New in version 2.6.*

2. The 'q' and 'Q' conversion codes are available in native mode only if the platform C compiler supports C long long, or, on Windows, _int64. They are always available in standard
modes.

*New in version 2.2.*

3. When attempting to pack a non-integer using any of the integer conversion codes, if the non-integer has a \_\_index\_() method then that method is called to convert the argument to an integer before packing. If no \_\_index\_() method exists, or the call to \_\_index\_() raises `TypeError`, then the \_\_int\_() method is tried. However, the use of \_\_int\_() is deprecated, and will raise `DeprecationWarning`.

*Changed in version 2.7:* Use of the \_\_index\_() method for non-integers is new in 2.7.

*Changed in version 2.7:* Prior to version 2.7, not all integer conversion codes would use the \_\_int\_() method to convert, and `DeprecationWarning` was raised only for float arguments.

4. For the \'f\' and \'d\' conversion codes, the packed representation uses the IEEE 754 binary32 (for \'f\') or binary64 (for \'d\') format, regardless of the floating-point format used by the platform.

5. The \'P\' format character is only available for the native byte ordering (selected as the default or with the \'@\' byte order character). The byte order character \='\' chooses to use little- or big-endian ordering based on the host system. The `struct` module does not interpret this as native ordering, so the \'P\' format is not available.

A format character may be preceded by an integral repeat count. For example, the format string \'4h\' means exactly the same as \'hhhh\'.

Whitespace characters between formats are ignored; a count and its format must not contain whitespace though.

For the 's' format character, the count is interpreted as the size of the string, not a repeat count like for the other format characters; for example, '10s' means a single 10-byte string, while '10c' means 10 characters. If a count is not given, it defaults to 1. For packing, the string is truncated or padded with null bytes as appropriate to make it fit. For unpacking, the resulting string always has exactly the specified number of bytes. As a special case, '0s' means a single, empty string (while '0c' means 0 characters).

The 'p' format character encodes a “Pascal string”, meaning a short variable-length string stored in a fixed number of bytes, given by the count. The first byte stored is the length of the string, or 255, whichever is smaller. The bytes of the string follow. If the string passed in to pack() is too long (longer than the count minus 1), only the leading count-1 bytes of the string are stored. If the string is shorter than count-1, it is padded with null bytes so that exactly count bytes in all are used. Note that for unpack(), the 'p' format character consumes count bytes, but that the string returned can never contain more than 255 characters.

For the 'P' format character, the return value is a Python integer or long integer, depending on the size needed to hold a pointer when it has been cast to an integer type. A NULL pointer will always be returned as the Python integer 0. When packing pointer-sized values, Python integer or long integer objects may be used. For example, the Alpha and Merced processors use 64-bit pointer values, meaning a Python long integer will be used to hold the pointer; other platforms use 32-bit pointers and will use a Python integer.

For the '?' format character, the return value is either True or False.
When packing, the truth value of the argument object is used. Either 0 or 1 in the native or standard bool representation will be packed, and any non-zero value will be True when unpacking.

### 7.3.2.3. Examples

**Note:** All examples assume a native byte order, size, and alignment with a big-endian machine.

A basic example of packing/unpacking three integers:

```python
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('hhl', '\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsizel('hhl')
8
```

Unpacked fields can be named by assigning them to variables or by wrapping the result in a named tuple:

```python
>>> record = 'raymond  \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)
>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name='raymond', serialnum=4658, school=264, gradelevel=8)
```

The ordering of format characters may have an impact on size since the padding needed to satisfy alignment requirements is different:

```python
>>> pack('ci', '*', 0x12131415)
'*\x00\x00\x00\x12\x13\x14\x15'
>>> pack('ic', 0x12131415, '*')
'\x12\x13\x14\x15*'
>>> calcsizel('ci')
8
```
The following format `'llh0l'` specifies two pad bytes at the end, assuming longs are aligned on 4-byte boundaries:

```
>>> pack('llh0l', 1, 2, 3)
'\x00\x00\x00\x10\x00\x00\x02\x00\x03\x00\x00'
```

This only works when native size and alignment are in effect; standard size and alignment does not enforce any alignment.

See also:

**Module array**
- Packed binary storage of homogeneous data.

**Module xdrlib**
- Packing and unpacking of XDR data.
7.3.3. Classes

The `struct` module also defines the following type:

```python
class struct.Struct(format)
```

Return a new Struct object which writes and reads binary data according to the format string `format`. Creating a Struct object once and calling its methods is more efficient than calling the `struct` functions with the same format since the format string only needs to be compiled once.

*New in version 2.5.*

Compiled Struct objects support the following methods and attributes:

- `pack(v1, v2, ...)`
  Identical to the `pack()` function, using the compiled format. 
  `(len(result)` will equal `self.size).`

- `pack_into(buffer, offset, v1, v2, ...)`
  Identical to the `pack_into()` function, using the compiled format.

- `unpack(string)`
  Identical to the `unpack()` function, using the compiled format. 
  `(len(string) must equal `self.size).`

- `unpack_from(buffer, offset=0)`
  Identical to the `unpack_from()` function, using the compiled format. `(len(buffer[offset:]) must be at least `self.size).`
The format string used to construct this Struct object.

**size**

The calculated size of the struct (and hence of the string) corresponding to *format*. 
7.4. difflib — Helpers for computing deltas

New in version 2.1.

This module provides classes and functions for comparing sequences. It can be used for example, for comparing files, and can produce difference information in various formats, including HTML and context and unified diffs. For comparing directories and files, see also, the filecmp module.

class difflib.SequenceMatcher
This is a flexible class for comparing pairs of sequences of any type, so long as the sequence elements are hashable. The basic algorithm predates, and is a little fancier than, an algorithm published in the late 1980’s by Ratcliff and Obershulp under the hyperbolic name “gestalt pattern matching.” The idea is to find the longest contiguous matching subsequence that contains no “junk” elements (the Ratcliff and Obershulp algorithm doesn’t address junk). The same idea is then applied recursively to the pieces of the sequences to the left and to the right of the matching subsequence. This does not yield minimal edit sequences, but does tend to yield matches that “look right” to people.

Timing: The basic Ratcliff-Obershulp algorithm is cubic time in the worst case and quadratic time in the expected case. SequenceMatcher is quadratic time for the worst case and has expected-case behavior dependent in a complicated way on how many elements the sequences have in common; best case time is linear.

Automatic junk heuristic: SequenceMatcher supports a heuristic
that automatically treats certain sequence items as junk. The heuristic counts how many times each individual item appears in the sequence. If an item’s duplicates (after the first one) account for more than 1% of the sequence and the sequence is at least 200 items long, this item is marked as “popular” and is treated as junk for the purpose of sequence matching. This heuristic can be turned off by setting the autojunk argument to False when creating the SequenceMatcher.

New in version 2.7.1: The autojunk parameter.

class difflib.Differ
This is a class for comparing sequences of lines of text, and producing human-readable differences or deltas. Differ uses SequenceMatcher both to compare sequences of lines, and to compare sequences of characters within similar (near-matching) lines.

Each line of a Differ delta begins with a two-letter code:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-'</td>
<td>line unique to sequence 1</td>
</tr>
<tr>
<td>'+'</td>
<td>line unique to sequence 2</td>
</tr>
<tr>
<td>'?'</td>
<td>line common to both sequences</td>
</tr>
<tr>
<td>'?'</td>
<td>line not present in either input sequence</td>
</tr>
</tbody>
</table>

Lines beginning with ‘?’ attempt to guide the eye to intraline differences, and were not present in either input sequence. These lines can be confusing if the sequences contain tab characters.

class difflib.HtmlDiff
This class can be used to create an HTML table (or a complete HTML file containing the table) showing a side by side, line by
line comparison of text with inter-line and intra-line change highlights. The table can be generated in either full or contextual difference mode.

The constructor for this class is:

```python
__init__(tabsize=8, wrapcolumn=None, linejunk=None, charjunk=IS_CHARACTER_JUNK)
```

Initializes instance of `HtmlDiff`.

* `tabsize` is an optional keyword argument to specify tab stop spacing and defaults to 8.

* `wrapcolumn` is an optional keyword to specify column number where lines are broken and wrapped, defaults to `None` where lines are not wrapped.

* `linejunk` and `charjunk` are optional keyword arguments passed into `ndiff()` (used by `HtmlDiff` to generate the side by side HTML differences). See `ndiff()` documentation for argument default values and descriptions.

The following methods are public:

```python
make_file(fromlines, tolines [, fromdesc][, todesc][, context][, numlines])
```

Compares `fromlines` and `tolines` (lists of strings) and returns a string which is a complete HTML file containing a table showing line by line differences with inter-line and intra-line changes highlighted.

* `fromdesc` and `todesc` are optional keyword arguments to specify from/to file column header strings (both default to an empty string).
context and numlines are both optional keyword arguments. Set context to True when contextual differences are to be shown, else the default is False to show the full files. numlines defaults to 5. When context is True numlines controls the number of context lines which surround the difference highlights. When context is False numlines controls the number of lines which are shown before a difference highlight when using the “next” hyperlinks (setting to zero would cause the “next” hyperlinks to place the next difference highlight at the top of the browser without any leading context).

```
make_table(fromlines, tolines [, fromdesc][, todesc][, context][, numlines])
```

Compares fromlines and tolines (lists of strings) and returns a string which is a complete HTML table showing line by line differences with inter-line and intra-line changes highlighted.

The arguments for this method are the same as those for the `make_file()` method.

Tools/scripts/diff.py is a command-line front-end to this class and contains a good example of its use.

New in version 2.4.

```
difflib.context_diff(a, b[, fromfile][, tofile][, fromfiledate][, tofiledate][, n][, lineterm])
```

Compare a and b (lists of strings); return a delta (a generator generating the delta lines) in context diff format.

Context diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in a before/after style. The number of context lines is set by n which defaults to three.
By default, the diff control lines (those with *** or ---) are created with a trailing newline. This is helpful so that inputs created from `file.readlines()` result in diffs that are suitable for use with `file.writelines()` since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the `lineterm` argument to "" so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for `fromfile`, `tofile`, `fromfiledate`, and `tofiledate`. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```python
>>> s1 = ['bacon
', 'eggs
', 'ham
', 'guido
']
>>> s2 = ['python
', 'eggy
', 'hamster
', 'guido
']
>>> for line in context_diff(s1, s2, fromfile='before.py', t...
...    sys.stdout.write(line)
*** before.py
--- after.py
***************
*** 1,4 ****
! bacon
! eggs
! ham
! guido
--- 1,4 ----
! python
! eggy
! hamster
! guido
```

See A command-line interface to difflib for a more detailed example.

*New in version 2.3.*

difflib.get_close_matches(word, possibilities[, n][, cutoff])
Return a list of the best “good enough” matches. \textit{word} is a sequence for which close matches are desired (typically a string), and \textit{possibilities} is a list of sequences against which to match \textit{word} (typically a list of strings).

Optional argument \textit{n} (default 3) is the maximum number of close matches to return; \textit{n} must be greater than 0.

Optional argument \textit{cutoff} (default 0.6) is a float in the range \([0, 1]\). Possibilities that don’t score at least that similar to \textit{word} are ignored.

The best (no more than \textit{n}) matches among the possibilities are returned in a list, sorted by similarity score, most similar first.

\begin{verbatim}
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy']
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('apple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
\end{verbatim}

\texttt{difflib.ndiff}(a, b[, \texttt{linejunk}][, \texttt{charjunk}])

Compare \texttt{a} and \texttt{b} (lists of strings); return a \texttt{Differ}-style delta (a generator generating the delta lines).

Optional keyword parameters \texttt{linejunk} and \texttt{charjunk} are for filter functions (or \texttt{None}):

\texttt{linejunk}: A function that accepts a single string argument, and returns true if the string is junk, or false if not. The default is (\texttt{None}), starting with Python 2.3. Before then, the default was the module-level function \texttt{IS_LINE_JUNK()}, which filters out lines
without visible characters, except for at most one pound character ('#'). As of Python 2.3, the underlying `SequenceMatcher` class does a dynamic analysis of which lines are so frequent as to constitute noise, and this usually works better than the pre-2.3 default.

`charjunk`: A function that accepts a character (a string of length 1), and returns if the character is junk, or false if not. The default is module-level function `IS_CHARACTER_JUNK()`, which filters out whitespace characters (a blank or tab; note: bad idea to include newline in this!).

Tools/scripts/ndiff.py is a command-line front-end to this function.

```python
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(1),
               'ore\ntree\nemu\n'.splitlines(1))
>>> print ''.join(diff),
- one
  ? ^
+ ore
  ? ^
- two
- three
  ? -
+ tree
+ emu
```

difflib. `restore(sequence, which)`

Return one of the two sequences that generated a delta.

Given a `sequence` produced by `Differ.compare()` or `ndiff()`, extract lines originating from file 1 or 2 (parameter `which`), stripping off line prefixes.

Example:

```python
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(1),
               'ore\ntree\nemu\n'.splitlines(1))
```
difflib.unified_diff(a, b[, fromfile][, tofile][, fromfiledate][, tofiledate][, n][, lineterm])

Compare a and b (lists of strings); return a delta (a generator
generating the delta lines) in unified diff format.

Unified diffs are a compact way of showing just the lines that
have changed plus a few lines of context. The changes are
shown in an inline style (instead of separate before/after blocks).
The number of context lines is set by n which defaults to three.

By default, the diff control lines (those with ---, +++, or @@) are
created with a trailing newline. This is helpful so that inputs
created from file.readlines() result in diffs that are suitable for
use with file.writelines() since both the inputs and outputs
have trailing newlines.

For inputs that do not have trailing newlines, set the lineterm
argument to "" so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and
modification times. Any or all of these may be specified using
strings for fromfile, tofile, fromfiledate, and tofiledate. The
modification times are normally expressed in the ISO 8601
format. If not specified, the strings default to blanks.

>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
See A command-line interface to difflib for a more detailed example.

New in version 2.3.

difflib.\texttt{IS\_LINE\_JUNK}(line)

Return true for ignorable lines. The line line is ignorable if line is blank or contains a single '\#', otherwise it is not ignorable. Used as a default for parameter linejunk in ndiff() before Python 2.3.

difflib.\texttt{IS\_CHARACTER\_JUNK}(ch)

Return true for ignorable characters. The character ch is ignorable if ch is a space or tab, otherwise it is not ignorable. Used as a default for parameter charjunk in ndiff().

See also:

Pattern Matching: The Gestalt Approach

Discussion of a similar algorithm by John W. Ratcliff and D. E. Metzener. This was published in Dr. Dobb's Journal in July, 1988.
SequenceMatcher Objects

The `SequenceMatcher` class has this constructor:

```python
class difflib.SequenceMatcher(isjunk=None, a='', b='', autojunk=True)
```

Optional argument `isjunk` must be `None` (the default) or a one-argument function that takes a sequence element and returns true if and only if the element is “junk” and should be ignored. Passing `None` for `isjunk` is equivalent to passing `lambda x: 0`; in other words, no elements are ignored. For example, pass:

```python
lambda x: x in "\t"
```

if you’re comparing lines as sequences of characters, and don’t want to synch up on blanks or hard tabs.

The optional arguments `a` and `b` are sequences to be compared; both default to empty strings. The elements of both sequences must be `hashable`.

The optional argument `autojunk` can be used to disable the automatic junk heuristic.

*New in version 2.7.1:* The `autojunk` parameter.

`SequenceMatcher` objects have the following methods:

```python
set_seqs(a, b)
```

Set the two sequences to be compared.

`SequenceMatcher` computes and caches detailed information about the second sequence, so if you want to compare one sequence against many sequences, use `set_seq2()` to set the
commonly used sequence once and call \texttt{set\_seq1()} repeatedly, once for each of the other sequences.

\textbf{set\_seq1(a)}

Set the first sequence to be compared. The second sequence to be compared is not changed.

\textbf{set\_seq2(b)}

Set the second sequence to be compared. The first sequence to be compared is not changed.

\textbf{find\_longest\_match(alo, ahi, blo, bhi)}

Find longest matching block in \texttt{a[alo:ahi]} and \texttt{b[blo:bhi]}.

If \texttt{isjunk} was omitted or \texttt{None}, \texttt{find\_longest\_match()} returns \( (i, j, k) \) such that \texttt{a[i:i+k]} is equal to \texttt{b[j:j+k]}, where \texttt{alo} \( \leq i \leq i+k \leq ahi \) and \texttt{blo} \( \leq j \leq j+k \leq bhî}. For all \( (i', j', k') \) meeting those conditions, the additional conditions \( k \geq k' \), \( i \leq i' \), and if \( i = i' \), \( j \leq j' \) are also met. In other words, of all maximal matching blocks, return one that starts earliest in \( a \), and of all those maximal matching blocks that start earliest in \( a \), return the one that starts earliest in \( b \).

```python
>>> s = SequenceMatcher(None, "abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

If \texttt{isjunk} was provided, first the longest matching block is determined as above, but with the additional restriction that no junk element appears in the block. Then that block is extended as far as possible by matching (only) junk elements on both sides. So the resulting block never matches on junk except as identical junk happens to be adjacent to an interesting match.
Here's the same example as before, but considering blanks to be junk. That prevents 'abcd' from matching the 'abcd' at the tail end of the second sequence directly. Instead only the 'abcd' can match, and matches the leftmost 'abcd' in the second sequence:

```python
>>> s = SequenceMatcher(lambda x: x==' ', 'abcd', 'abcd
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

If no blocks match, this returns (alo, blo, 0).

*Changed in version 2.6*: This method returns a named tuple

```
Match(a, b, size)
```

**get_matching_blocks()**

Return list of triples describing matching subsequences. Each triple is of the form (i, j, n), and means that a[i:i+n] == b[j:j+n]. The triples are monotonically increasing in i and j.

The last triple is a dummy, and has the value (len(a), len(b), 0). It is the only triple with n == 0. If (i, j, n) and (i', j', n') are adjacent triples in the list, and the second is not the last triple in the list, then i+n != i' or j+n != j'; in other words, adjacent triples always describe non-adjacent equal blocks.

*Changed in version 2.5*: The guarantee that adjacent triples always describe non-adjacent blocks was implemented.

```python
>>> s = SequenceMatcher(None, 'abxcd', 'abcd')
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(]
```

```python
```
**get_opcodes()**

Return list of 5-tuples describing how to turn $a$ into $b$. Each tuple is of the form $(tag, i_1, i_2, j_1, j_2)$. The first tuple has $i_1 == j_1 == 0$, and remaining tuples have $i_1$ equal to the $i_2$ from the preceding tuple, and, likewise, $j_1$ equal to the previous $j_2$.

The *tag* values are strings, with these meanings:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'replace'</td>
<td>$a[i_1:i_2]$ should be replaced by $b[j_1:j_2]$.</td>
</tr>
<tr>
<td>'delete'</td>
<td>$a[i_1:i_2]$ should be deleted. Note that $j_1 == j_2$ in this case.</td>
</tr>
<tr>
<td>'insert'</td>
<td>$b[j_1:j_2]$ should be inserted at $a[i_1:i_1]$. Note that $i_1 == i_2$ in this case.</td>
</tr>
<tr>
<td>'equal'</td>
<td>$a[i_1:i_2] == b[j_1:j_2]$ (the sub-sequences are equal).</td>
</tr>
</tbody>
</table>

For example:

```python
>>> a = "qabxcd"
>>> b = "abyxcd"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print("%7s %s[%d:%d] (%s) %s[%d:%d] (%s)" %
...             (tag, i1, i2, a[i1:i2], j1, j2, b[j1:j2]))
          delete a[0:1] (q) b[0:0] ()
          equal a[1:3] (ab) b[0:2] (ab)
          replace a[3:4] (x) b[2:3] (y)
```

**get_grouped_opcodes([n])**

Return a *generator* of groups with up to $n$ lines of context.

Starting with the groups returned by *get_opcodes()*, this method splits out smaller change clusters and eliminates intervening ranges which have no changes.
The groups are returned in the same format as get_opcodes().

New in version 2.3.

**ratio()**

Return a measure of the sequences' similarity as a float in the range [0, 1].

Where T is the total number of elements in both sequences, and M is the number of matches, this is 2.0*M / T. Note that this is 1.0 if the sequences are identical, and 0.0 if they have nothing in common.

This is expensive to compute if get_matching_blocks() or get_opcodes() hasn't already been called, in which case you may want to try quick_ratio() or real_quick_ratio() first to get an upper bound.

**quick_ratio()**

Return an upper bound on ratio() relatively quickly.

**real_quick_ratio()**

Return an upper bound on ratio() very quickly.

The three methods that return the ratio of matching to total characters can give different results due to differing levels of approximation, although quick_ratio() and real_quick_ratio() are always at least as large as ratio():

```python
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
0.75
```
7.4.2. SequenceMatcher Examples

This example compares two strings, considering blanks to be “junk:"

```python
>>> s = SequenceMatcher(lambda x: x == " ",
                       "private Thread currentThread;",
                       "private volatile Thread currentThread;"
)
```

`ratio()` returns a float in [0, 1], measuring the similarity of the sequences. As a rule of thumb, a `ratio()` value over 0.6 means the sequences are close matches:

```python
>>> print round(s.ratio(), 3)
0.866
```

If you're only interested in where the sequences match, `get_matching_blocks()` is handy:

```python
>>> for block in s.get_matching_blocks():
...     print "a[%d] and b[%d] match for %d elements" % block
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

Note that the last tuple returned by `get_matching_blocks()` is always a dummy, `(len(a), len(b), 0)`, and this is the only case in which the last tuple element (number of elements matched) is 0.

If you want to know how to change the first sequence into the second, use `get_opcodes()`:

```python
>>> for opcode in s.get_opcodes():
...     print "%s a[%d:%d] b[%d:%d]" % opcode
equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
equal a[8:29] b[17:38]
```
See also:

- The `get_close_matches()` function in this module which shows how simple code building on `SequenceMatcher` can be used to do useful work.
- Simple version control recipe for a small application built with `SequenceMatcher`.
7.4.3. Differ Objects

Note that Differ-generated deltas make no claim to be minimal diffs. To the contrary, minimal diffs are often counter-intuitive, because they synch up anywhere possible, sometimes accidental matches 100 pages apart. Restricting synch points to contiguous matches preserves some notion of locality, at the occasional cost of producing a longer diff.

The Differ class has this constructor:

```python
class difflib.Differ([linejunk[, charjunk]])
```

Optional keyword parameters linejunk and charjunk are for filter functions (or None):

- **linejunk**: A function that accepts a single string argument, and returns true if the string is junk. The default is None, meaning that no line is considered junk.

- **charjunk**: A function that accepts a single character argument (a string of length 1), and returns true if the character is junk. The default is None, meaning that no character is considered junk.

Differ objects are used (deltas generated) via a single method:

```python
compare(a, b)
```

Compare two sequences of lines, and generate the delta (a sequence of lines).

Each sequence must contain individual single-line strings ending with newlines. Such sequences can be obtained from the `readlines()` method of file-like objects. The delta generated also consists of newline-terminated strings, ready
to be printed as-is via the `writelines()` method of a file-like object.
This example compares two texts. First we set up the texts, sequences of individual single-line strings ending with newlines (such sequences can also be obtained from the `readlines()` method of file-like objects):

```python
>>> text1 = ''' 1. Beautiful is better than ugly. ...
   2. Explicit is better than implicit.
   ... 3. Simple is better than complex.
   ... 4. Complex is better than complicated.
   ... '''.splitlines(1)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = ''' 1. Beautiful is better than ugly.
   ... 3. Simple is better than complex.
   ... 4. Complicated is better than complex.
   ... 5. Flat is better than nested.
   ... '''.splitlines(1)
```

Next we instantiate a Differ object:

```python
>>> d = Differ()
```

Note that when instantiating a Differ object we may pass functions to filter out line and character “junk.” See the `Differ()` constructor for details.

Finally, we compare the two:

```python
>>> result = list(d.compare(text1, text2))
```

`result` is a list of strings, so let’s pretty-print it:

```python
>>> from pprint import pprint
```
>>> pprint(result)
['  1. Beautiful is better than ugly.
', '
', '-  2. Explicit is better than implicit.
', '
', '-  3. Simple is better than complex.
', '+  3. Simple is better than complex.
', '?  ++
', '-  4. Complex is better than complicated.
', '?  ^    ---- ^
', '+  4. Complicated is better than complex.
', '?  ++++ ^
', '+  5. Flat is better than nested.
]

As a single multi-line string it looks like this:

>>> import sys
>>> sys.stdout.writelines(result)
1. Beautiful is better than ugly.
- 2. Explicit is better than implicit.
- 3. Simple is better than complex.
+ 3. Simple is better than complex.
?  ++
- 4. Complex is better than complicated.
?  ^    ---- ^
+ 4. Complicated is better than complex.
?  ++++ ^
+ 5. Flat is better than nested.
This example shows how to use difflib to create a diff-like utility. It is also contained in the Python source distribution, as Tools/scripts/diff.py.

""" Command line interface to difflib.py providing diffs in four formats:
* ndiff: lists every line and highlights interline changes.
* context: highlights clusters of changes in a before/after format.
* unified: highlights clusters of changes in an inline format.
* html: generates side by side comparison with change highlights.
"""

import sys, os, time, difflib, optparse

def main():
    # Configure the option parser
    usage = "usage: %prog [options] fromfile tofile"
    parser = optparse.OptionParser(usage)
    parser.add_option("-c", action="store_true", default=False,
                      help='Produce a context format diff (default)'
                      )
    parser.add_option("-u", action="store_true", default=False,
                      help='Produce a unified format diff'
                      )
    hlp = 'Produce HTML side by side diff (can use -c and -l in
    parser.add_option("-m", action="store_true", default=False,
    parser.add_option("-n", action="store_true", default=False,
                      help='Produce a ndiff format diff'
                      )
    parser.add_option("-l", "--lines", type="int", default=3,
                      help='Set number of context lines (default 3)
    (options, args) = parser.parse_args()

    if len(args) == 0:
        parser.print_help()
        sys.exit(1)
    if len(args) != 2:
        parser.error("need to specify both a fromfile and tofile")

    n = options.lines
    fromfile, tofile = args  # as specified in the usage string
    # we're passing these as arguments to the diff function
fromdate = time.ctime(os.stat(fromfile).st_mtime)
todate = time.ctime(os.stat(tofile).st_mtime)
fromlines = open(fromfile, 'U').readlines()
tolines = open(tofile, 'U').readlines()

if options.u:
    diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile, fromdate, todate, n=n)
elif options.n:
    diff = difflib.ndiff(fromlines, tolines)
elif options.m:
    diff = difflib.HtmlDiff().make_file(fromlines, tolines, tofile, n=numlines=n)
else:
    diff = difflib.context_diff(fromlines, tolines, fromfile, tofile, fromdate, todate, n=n)

# we're using writelines because diff is a generator
sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()
7.5. `StringIO` — Read and write strings as files

This module implements a file-like class, `StringIO`, that reads and writes a string buffer (also known as `memory files`). See the description of file objects for operations (section File Objects). (For standard strings, see `str` and `unicode`.)

```python
class StringIO:

    def __init__(self, buffer=None):
        # Implementation details...
```

When a `StringIO` object is created, it can be initialized to an existing string by passing the string to the constructor. If no string is given, the `StringIO` will start empty. In both cases, the initial file position starts at zero.

The `StringIO` object can accept either Unicode or 8-bit strings, but mixing the two may take some care. If both are used, 8-bit strings that cannot be interpreted as 7-bit ASCII (that use the 8th bit) will cause a `UnicodeError` to be raised when `.getvalue()` is called.

The following methods of `StringIO` objects require special mention:

`StringIO.getvalue()`

Retrieve the entire contents of the “file” at any time before the `StringIO` object’s `close()` method is called. See the note above for information about mixing Unicode and 8-bit strings; such mixing can cause this method to raise `UnicodeError`.

`StringIO.close()`

Free the memory buffer. Attempting to do further operations with a closed `StringIO` object will raise a `ValueError`. 
Example usage:

```python
import StringIO

output = StringIO.StringIO()
output.write('First line.\n')
print >>output, 'Second line.'

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# ..getvalue() will now raise an exception.
output.close()
```
The module `cStringIO` provides an interface similar to that of the `StringIO` module. Heavy use of `StringIO.StringIO` objects can be made more efficient by using the function `StringIO()` from this module instead.

`cStringIO.StringIO([s])`
Return a `StringIO`-like stream for reading or writing.

Since this is a factory function which returns objects of built-in types, there's no way to build your own version using subclassing. It's not possible to set attributes on it. Use the original `StringIO` module in those cases.

Unlike the `StringIO` module, this module is not able to accept Unicode strings that cannot be encoded as plain ASCII strings.

Another difference from the `StringIO` module is that calling `StringIO()` with a string parameter creates a read-only object. Unlike an object created without a string parameter, it does not have write methods. These objects are not generally visible. They turn up in tracebacks as `StringI` and `StringO`.

The following data objects are provided as well:

`cStringIO.InputType`
The type object of the objects created by calling `StringIO()` with a string parameter.

`cStringIO.OutputType`
The type object of the objects returned by calling `StringIO()` with
no parameters.

There is a C API to the module as well; refer to the module source for more information.

Example usage:

```python
import cStringIO
output = cStringIO.StringIO()
output.write('First line.
')
print >>output, 'Second line.'

# Retrieve file contents -- this will be
# 'First line.
Second line.
'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```
7.7. `textwrap` — Text wrapping and filling

*New in version 2.3.*

**Source code:** `Lib/textwrap.py`

The `textwrap` module provides two convenience functions, `wrap()` and `fill()`, as well as `TextWrapper`, the class that does all the work, and a utility function `dedent()`. If you're just wrapping or filling one or two text strings, the convenience functions should be good enough; otherwise, you should use an instance of `TextWrapper` for efficiency.

```python
import textwrap

textwrap.wrap(text[, width[, ...]])
    Wraps the single paragraph in `text` (a string) so every line is at most `width` characters long. Returns a list of output lines, without final newlines.

    Optional keyword arguments correspond to the instance attributes of `TextWrapper`, documented below. `width` defaults to 70.

    See the `TextWrapper.wrap()` method for additional details on how `wrap()` behaves.
```

textwrap.fill(text[, width[, ...]])
    Wraps the single paragraph in `text`, and returns a single string containing the wrapped paragraph. `fill()` is shorthand for

```
"\n".join(wrap(text, ...))
```
In particular, \texttt{fill()} accepts exactly the same keyword arguments as \texttt{wrap()}.

Both \texttt{wrap()} and \texttt{fill()} work by creating a \texttt{TextWrapper} instance and calling a single method on it. That instance is not reused, so for applications that wrap/fill many text strings, it will be more efficient for you to create your own \texttt{TextWrapper} object.

Text is preferably wrapped on whitespaces and right after the hyphens in hyphenated words; only then will long words be broken if necessary, unless \texttt{TextWrapper.break_long_words} is set to false.

An additional utility function, \texttt{dedent()}, is provided to remove indentation from strings that have unwanted whitespace to the left of the text.

\texttt{textwrap.dedent(text)}

Remove any common leading whitespace from every line in \textit{text}.

This can be used to make triple-quoted strings line up with the left edge of the display, while still presenting them in the source code in indented form.

Note that tabs and spaces are both treated as whitespace, but they are not equal: the lines "hello" and "\thello" are considered to have no common leading whitespace. (This behaviour is new in Python 2.5; older versions of this module incorrectly expanded tabs before searching for common leading whitespace.)

For example:

\begin{python}
def test():
    # end first line with \ to avoid the empty line!
    s = '''\n    hello
\end{python}
class textwrap.TextWrapper(...)

The TextWrapper constructor accepts a number of optional keyword arguments. Each argument corresponds to one instance attribute, so for example

```python
wrapper = TextWrapper(initial_indent="* ")
```

is the same as

```python
wrapper = TextWrapper()
wrapper.initial_indent = "* 
```

You can re-use the same TextWrapper object many times, and you can change any of its options through direct assignment to instance attributes between uses.

The TextWrapper instance attributes (and keyword arguments to the constructor) are as follows:

**width**

(default: 70) The maximum length of wrapped lines. As long as there are no individual words in the input text longer than width, TextWrapper guarantees that no output line will be longer than width characters.

**expand_tabs**

(default: True) If true, then all tab characters in text will be expanded to spaces using the expandtabs() method of text.

**replace_whitespace**

(default: True) If true, after tab expansion but before
wrapping, the \texttt{wrap()} method will replace each whitespace character with a single space. The whitespace characters replaced are as follows: tab, newline, vertical tab, formfeed, and carriage return ("\t\n\v\f\r").

\textbf{Note:} If \texttt{expand\_tabs} is false and \texttt{replace\_whitespace} is true, each tab character will be replaced by a single space, which is \textit{not} the same as tab expansion.

\textbf{Note:} If \texttt{replace\_whitespace} is false, newlines may appear in the middle of a line and cause strange output. For this reason, text should be split into paragraphs (using \texttt{str.splitlines()} or similar) which are wrapped separately.

drop\_whitespace
(default: \texttt{True}) If true, whitespace at the beginning and ending of every line (after wrapping but before indenting) is dropped. Whitespace at the beginning of the paragraph, however, is not dropped if non-whitespace follows it. If whitespace being dropped takes up an entire line, the whole line is dropped.

\textit{New in version 2.6:} Whitespace was always dropped in earlier versions.

initial\_indent
(default: \texttt{''}) String that will be prepended to the first line of wrapped output. Counts towards the length of the first line. The empty string is not indented.

subsequent\_indent
(default: \texttt{''}) String that will be prepended to all lines of wrapped output except the first. Counts towards the length of each line except the first.

fix\_sentence\_endings
If true, `TextWrapper` attempts to detect sentence endings and ensure that sentences are always separated by exactly two spaces. This is generally desired for text in a monospaced font. However, the sentence detection algorithm is imperfect: it assumes that a sentence ending consists of a lowercase letter followed by one of ".", ",", or "?", possibly followed by one of "" or """, followed by a space. One problem with this is algorithm is that it is unable to detect the difference between “Dr.” in

```plaintext
[...] Dr. Frankenstein's monster [...]  
```

and “Spot.” in

```plaintext
[...] See Spot. See Spot run [...]  
```

`fix_sentence_endings` is false by default.

Since the sentence detection algorithm relies on `string.lowercase` for the definition of “lowercase letter,” and a convention of using two spaces after a period to separate sentences on the same line, it is specific to English-language texts.

**break_long_words**

(default: `False`) If true, then words longer than `width` will be broken in order to ensure that no lines are longer than `width`. If it is false, long words will not be broken, and some lines may be longer than `width`. (Long words will be put on a line by themselves, in order to minimize the amount by which `width` is exceeded.)

**break_on_hyphens**

(default: `True`) If true, wrapping will occur preferably on whitespaces and right after hyphens in compound words, as it
is customary in English. If false, only whitespaces will be considered as potentially good places for line breaks, but you need to set `break_long_words` to false if you want truly inseparable words. Default behaviour in previous versions was to always allow breaking hyphenated words.

*New in version 2.6.*

**TextWrapper** also provides two public methods, analogous to the module-level convenience functions:

**wrap**(text)
- Wraps the single paragraph in `text` (a string) so every line is at most `width` characters long. All wrapping options are taken from instance attributes of the **TextWrapper** instance. Returns a list of output lines, without final newlines. If the wrapped output has no content, the returned list is empty.

**fill**(text)
- Wraps the single paragraph in `text`, and returns a single string containing the wrapped paragraph.
7.8. codecs — Codec registry and base classes

This module defines base classes for standard Python codecs (encoders and decoders) and provides access to the internal Python codec registry which manages the codec and error handling lookup process.

It defines the following functions:

\[
\text{codecs.} \quad \text{encode(obj[, encoding[, errors]])}
\]

Encodes \( \text{obj} \) using the codec registered for \( \text{encoding} \). The default encoding is 'ascii'.

*Errors* may be given to set the desired error handling scheme. The default error handler is 'strict' meaning that encoding errors raise \( \text{ValueError} \) (or a more codec specific subclass, such as \( \text{UnicodeEncodeError} \)). Refer to Codec Base Classes for more information on codec error handling.

New in version 2.4.

\[
\text{codecs.} \quad \text{decode(obj[, encoding[, errors]])}
\]

Decodes \( \text{obj} \) using the codec registered for \( \text{encoding} \). The default encoding is 'ascii'.

*Errors* may be given to set the desired error handling scheme. The default error handler is 'strict' meaning that decoding errors raise \( \text{ValueError} \) (or a more codec specific subclass, such as \( \text{UnicodeDecodeError} \)). Refer to Codec Base Classes for more information on codec error handling.
New in version 2.4.

codecs.register(search_function)

Register a codec search function. Search functions are expected to take one argument, the encoding name in all lower case letters, and return a CodecInfo object having the following attributes:

- **name** The name of the encoding;
- **encode** The stateless encoding function;
- **decode** The stateless decoding function;
- **incrementalencoder** An incremental encoder class or factory function;
- **incrementaldecoder** An incremental decoder class or factory function;
- **streamwriter** A stream writer class or factory function;
- **streamreader** A stream reader class or factory function.

The various functions or classes take the following arguments:

- **encode** and **decode**: These must be functions or methods which have the same interface as the `encode()`/`decode()` methods of Codec instances (see Codec Interface). The functions/methods are expected to work in a stateless mode.

- **incrementalencoder** and **incrementaldecoder**: These have to be factory functions providing the following interface:

  ```python
  factory(errors='strict')
  ```

The factory functions must return objects providing the interfaces defined by the base classes IncrementalEncoder and IncrementalDecoder, respectively. Incremental codecs can maintain state.
streamreader and streamwriter: These have to be factory functions providing the following interface:

    factory(stream, errors='strict')

The factory functions must return objects providing the interfaces defined by the base classes StreamReader and StreamWriter, respectively. Stream codecs can maintain state.

Possible values for errors are

- 'strict': raise an exception in case of an encoding error
- 'replace': replace malformed data with a suitable replacement marker, such as '?' or '�'
- 'ignore': ignore malformed data and continue without further notice
- 'xmlcharrefreplace': replace with the appropriate XML character reference (for encoding only)
- 'backslashreplace': replace with backslashed escape sequences (for encoding only)

as well as any other error handling name defined via register_error().

In case a search function cannot find a given encoding, it should return None.

codecs. lookup(encoding)

Looks up the codec info in the Python codec registry and returns a CodecInfo object as defined above.

Encodings are first looked up in the registry's cache. If not found, the list of registered search functions is scanned. If no CodecInfo object is found, a LookupError is raised. Otherwise, the CodecInfo object is stored in the cache and returned to the caller.
To simplify access to the various codecs, the module provides these additional functions which use `lookup()` for the codec lookup:

```python
codecs.getencoder(encoding)

Look up the codec for the given encoding and return its encoder function.

Raises a `LookupError` in case the encoding cannot be found.
```

codecs.getdecoder(encoding)

Look up the codec for the given encoding and return its decoder function.

Raises a `LookupError` in case the encoding cannot be found.

```python
codecs.getincrementalencoder(encoding)

Look up the codec for the given encoding and return its incremental encoder class or factory function.

Raises a `LookupError` in case the encoding cannot be found or the codec doesn’t support an incremental encoder.

New in version 2.5.
```

codecs.getincrementaldecoder(encoding)

Look up the codec for the given encoding and return its incremental decoder class or factory function.

Raises a `LookupError` in case the encoding cannot be found or the codec doesn’t support an incremental decoder.

New in version 2.5.

```python
codecs.getreader(encoding)

Look up the codec for the given encoding and return its StreamReader class or factory function.
```
Raises a `LookupError` in case the encoding cannot be found.

codecs. `getwriter(encoding)`
Look up the codec for the given encoding and return its StreamWriter class or factory function.

Raises a `LookupError` in case the encoding cannot be found.

codecs. `register_error(name, error_handler)`
Register the error handling function `error_handler` under the name `name`. `error_handler` will be called during encoding and decoding in case of an error, when `name` is specified as the errors parameter.

For encoding `error_handler` will be called with a `UnicodeEncodeError` instance, which contains information about the location of the error. The error handler must either raise this or a different exception or return a tuple with a replacement for the unencodable part of the input and a position where encoding should continue. The encoder will encode the replacement and continue encoding the original input at the specified position. Negative position values will be treated as being relative to the end of the input string. If the resulting position is out of bound an `IndexError` will be raised.

Decoding and translating works similar, except `UnicodeDecodeError` or `UnicodeTranslateError` will be passed to the handler and that the replacement from the error handler will be put into the output directly.

codecs. `lookup_error(name)`
Return the error handler previously registered under the name `name`.

Raises a `LookupError` in case the handler cannot be found.
**codecs.** `strict_errors(exception)`

Implements the `strict` error handling: each encoding or decoding error raises a `UnicodeError`.

**codecs.** `replace_errors(exception)`

Implements the `replace` error handling: malformed data is replaced with a suitable replacement character such as `'?` in bytestrings and `\ufffd` in Unicode strings.

**codecs.** `ignore_errors(exception)`

Implements the `ignore` error handling: malformed data is ignored and encoding or decoding is continued without further notice.

**codecs.** `xmlcharrefreplace_errors(exception)`

Implements the `xmlcharrefreplace` error handling (for encoding only): the unencodable character is replaced by an appropriate XML character reference.

**codecs.** `backslashreplace_errors(exception)`

Implements the `backslashreplace` error handling (for encoding only): the unencodable character is replaced by a backslashed escape sequence.

To simplify working with encoded files or stream, the module also defines these utility functions:

**codecs.** `open(filename, mode[, encoding[, errors[, buffering]]]])`

Open an encoded file using the given `mode` and return a wrapped version providing transparent encoding/decoding. The default file mode is `'r'` meaning to open the file in read mode.

**Note:** The wrapped version will only accept the object format defined by the codecs, i.e. Unicode objects for most built-in codecs. Output is also codec-dependent and will usually be
Unicode as well.

**Note:** Files are always opened in binary mode, even if no binary mode was specified. This is done to avoid data loss due to encodings using 8-bit values. This means that no automatic conversion of '\n' is done on reading and writing.

*encoding* specifies the encoding which is to be used for the file.

*errors* may be given to define the error handling. It defaults to ['strict'] which causes a `ValueError` to be raised in case an encoding error occurs.

*buffering* has the same meaning as for the built-in `open()` function. It defaults to line buffered.

codecs.**EncodedFile**(file, input[, output[, errors]])

Return a wrapped version of file which provides transparent encoding translation.

Strings written to the wrapped file are interpreted according to the given *input* encoding and then written to the original file as strings using the *output* encoding. The intermediate encoding will usually be Unicode but depends on the specified codecs.

If *output* is not given, it defaults to *input*.

*errors* may be given to define the error handling. It defaults to ['strict'], which causes `ValueError` to be raised in case an encoding error occurs.

codecs.**iterencode**(iterable, encoding[, errors])

Uses an incremental encoder to iteratively encode the input provided by *iterable*. This function is a generator. *errors* (as well as any other keyword argument) is passed through to the
incremental encoder.

*New in version 2.5.*

codecs. **iterdecode**(iterable, encoding[, errors])

Uses an incremental decoder to iteratively decode the input provided by `iterable`. This function is a generator. `errors` (as well as any other keyword argument) is passed through to the incremental decoder.

*New in version 2.5.*

The module also provides the following constants which are useful for reading and writing to platform dependent files:

codecs. **BOM**
codecs. **BOM_BE**
codecs. **BOM_LE**
codecs. **BOM_UTF8**
codecs. **BOM_UTF16**
codecs. **BOM_UTF16_BE**
codecs. **BOM_UTF16_LE**
codecs. **BOM_UTF32**
codecs. **BOM_UTF32_BE**
codecs. **BOM_UTF32_LE**

These constants define various encodings of the Unicode byte order mark (BOM) used in UTF-16 and UTF-32 data streams to indicate the byte order used in the stream or file and in UTF-8 as a Unicode signature. `BOM_UTF16` is either `BOM_UTF16_BE` or `BOM_UTF16_LE` depending on the platform's native byte order, `BOM` is an alias for `BOM_UTF16`, `BOM_LE` for `BOM_UTF16_LE` and `BOM_BE` for `BOM_UTF16_BE`. The others represent the BOM in UTF-8 and UTF-32 encodings.
7.8.1. Codec Base Classes

The `codecs` module defines a set of base classes which define the interface and can also be used to easily write your own codecs for use in Python.

Each codec has to define four interfaces to make it usable as codec in Python: stateless encoder, stateless decoder, stream reader and stream writer. The stream reader and writers typically reuse the stateless encoder/decoder to implement the file protocols.

The `Codec` class defines the interface for stateless encoders/decoders.

To simplify and standardize error handling, the `encode()` and `decode()` methods may implement different error handling schemes by providing the `errors` string argument. The following string values are defined and implemented by all standard Python codecs:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'strict'</td>
<td>Raise <code>UnicodeError</code> (or a subclass); this is the default.</td>
</tr>
<tr>
<td>'ignore'</td>
<td>Ignore the character and continue with the next.</td>
</tr>
<tr>
<td>'replace'</td>
<td>Replace with a suitable replacement character; Python will use the official U+FFFD REPLACEMENT CHARACTER for the built-in Unicode codecs on decoding and ‘?’ on encoding.</td>
</tr>
<tr>
<td>'xmlcharrefreplace'</td>
<td>Replace with the appropriate XML character reference (only for encoding).</td>
</tr>
<tr>
<td>'backslashreplace'</td>
<td>Replace with backslashed escape sequences (only for encoding).</td>
</tr>
</tbody>
</table>
The set of allowed values can be extended via `register_error()`.

### 7.8.1.1. Codec Objects

The `codec` class defines these methods which also define the function interfaces of the stateless encoder and decoder:

**Codec. `encode(input[, errors])`**

Encodes the object `input` and returns a tuple `(output object, length consumed)`. While codecs are not restricted to use with Unicode, in a Unicode context, encoding converts a Unicode object to a plain string using a particular character set encoding (e.g., `cp1252` or `iso-8859-1`).

`errors` defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the `Codec` instance. Use `StreamWriter` for codecs which have to keep state in order to make encoding efficient.

The encoder must be able to handle zero length input and return an empty object of the output object type in this situation.

**Codec. `decode(input[, errors])`**

Decodes the object `input` and returns a tuple `(output object, length consumed)`. In a Unicode context, decoding converts a plain string encoded using a particular character set encoding to a Unicode object.

`input` must be an object which provides the `bf_getreadbuf` buffer slot. Python strings, buffer objects and memory mapped files are examples of objects providing this slot.
errors defines the error handling to apply. It defaults to 'strict' handling.

The method may not store state in the codec instance. Use StreamReader for codecs which have to keep state in order to make decoding efficient.

The decoder must be able to handle zero length input and return an empty object of the output object type in this situation.

The IncrementalEncoder and IncrementalDecoder classes provide the basic interface for incremental encoding and decoding. Encoding/decoding the input isn't done with one call to the stateless encoder/decoder function, but with multiple calls to the encode()/decode() method of the incremental encoder/decoder. The incremental encoder/decoder keeps track of the encoding/decoding process during method calls.

The joined output of calls to the encode()/decode() method is the same as if all the single inputs were joined into one, and this input was encoded/decoded with the stateless encoder/decoder.

### 7.8.1.2. IncrementalEncoder Objects

*New in version 2.5.*

The IncrementalEncoder class is used for encoding an input in multiple steps. It defines the following methods which every incremental encoder must define in order to be compatible with the Python codec registry.

```python
class codecs.IncrementalEncoder([errors])
    Constructor for an IncrementalEncoder instance.
```
All incremental encoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The IncrementalEncoder may implement different error handling schemes by providing the errors keyword argument. These parameters are predefined:

- 'strict' Raise ValueError (or a subclass); this is the default.
- 'ignore' Ignore the character and continue with the next.
- 'replace' Replace with a suitable replacement character
- 'xmlcharrefreplace' Replace with the appropriate XML character reference
- 'backslashreplace' Replace with backslashed escape sequences.

The errors argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the IncrementalEncoder object.

The set of allowed values for the errors argument can be extended with register_error().

encode(object[, final])

Encodes object (taking the current state of the encoder into account) and returns the resulting encoded object. If this is the last call to encode() final must be true (the default is false).

reset()

Reset the encoder to the initial state.
The `IncrementalDecoder` class is used for decoding an input in multiple steps. It defines the following methods which every incremental decoder must define in order to be compatible with the Python codec registry.

```python
class codecs.IncrementalDecoder([errors])
```

Constructor for an `IncrementalDecoder` instance.

All incremental decoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The `IncrementalDecoder` may implement different error handling schemes by providing the `errors` keyword argument. These parameters are predefined:

- `'strict'` Raise `ValueError` (or a subclass); this is the default.
- `'ignore'` Ignore the character and continue with the next.
- `'replace'` Replace with a suitable replacement character.

The `errors` argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `IncrementalDecoder` object.

The set of allowed values for the `errors` argument can be extended with `register_error()`.

```python
decode(object[, final])
```

Decodes `object` (taking the current state of the decoder into account) and returns the resulting decoded object. If this is
the last call to `decode()` `final` must be true (the default is false). If `final` is true the decoder must decode the input completely and must flush all buffers. If this isn't possible (e.g. because of incomplete byte sequences at the end of the input) it must initiate error handling just like in the stateless case (which might raise an exception).

```
reset()
```

Reset the decoder to the initial state.

The `StreamWriter` and `StreamReader` classes provide generic working interfaces which can be used to implement new encoding submodules very easily. See `encodings.utf_8` for an example of how this is done.

### 7.8.1.4. StreamWriter Objects

The `StreamWriter` class is a subclass of `Codec` and defines the following methods which every stream writer must define in order to be compatible with the Python codec registry.

```
class codecs.StreamWriter(stream[, errors])
```

Constructor for a `StreamWriter` instance.

All stream writers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

`stream` must be a file-like object open for writing binary data.

The `StreamWriter` may implement different error handling schemes by providing the `errors` keyword argument. These parameters are predefined:
- 'strict' Raise `ValueError` (or a subclass); this is the default.
- 'ignore' Ignore the character and continue with the next.
- 'replace' Replace with a suitable replacement character
- 'xmlcharrefreplace' Replace with the appropriate XML character reference
- 'backslashreplace' Replace with backslashed escape sequences.

The `errors` argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `StreamWriter` object.

The set of allowed values for the `errors` argument can be extended with `register_error()`.

`write(object)`

Writes the object's contents encoded to the stream.

`writelines(list)`

Writes the concatenated list of strings to the stream (possibly by reusing the `write()` method).

`reset()`

Flushes and resets the codec buffers used for keeping state.

Calling this method should ensure that the data on the output is put into a clean state that allows appending of new fresh data without having to rescan the whole stream to recover state.

In addition to the above methods, the `StreamWriter` must also inherit all other methods and attributes from the underlying stream.
7.8.1.5. StreamReader Objects

The **StreamReader** class is a subclass of **codec** and defines the following methods which every stream reader must define in order to be compatible with the Python codec registry.

```python
class codecs.StreamReader(stream[, errors])
```

Constructor for a **StreamReader** instance.

All stream readers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

*stream* must be a file-like object open for reading (binary) data.

The **StreamReader** may implement different error handling schemes by providing the *errors* keyword argument. These parameters are defined:

- **'strict'** Raise **ValueError** (or a subclass); this is the default.
- **'ignore'** Ignore the character and continue with the next.
- **'replace'** Replace with a suitable replacement character.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the **StreamReader** object.

The set of allowed values for the *errors* argument can be extended with **register_error()**.

```python
read([size[, chars[, firstline]]])
```

Decodes data from the stream and returns the resulting
object.

_chars_ indicates the number of characters to read from the stream. _read()_ will never return more than _chars_ characters, but it might return less, if there are not enough characters available.

_size_ indicates the approximate maximum number of bytes to read from the stream for decoding purposes. The decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. _size_ is intended to prevent having to decode huge files in one step.

_firstline_ indicates that it would be sufficient to only return the first line, if there are decoding errors on later lines.

The method should use a greedy read strategy meaning that it should read as much data as is allowed within the definition of the encoding and the given size, e.g. if optional encoding endings or state markers are available on the stream, these should be read too.

*Changed in version 2.4*: _chars_ argument added.

*Changed in version 2.4.2*: _firstline_ argument added.

```python
readline([size[, keepends]])
```

Read one line from the input stream and return the decoded data.

_size_, if given, is passed as size argument to the stream's _read()_ method.

If _keepends_ is false line-endings will be stripped from the lines returned.
Changed in version 2.4: keepends argument added.

**readlines**([`sizehint`, `keepends`])
Read all lines available on the input stream and return them as a list of lines.

Line-endings are implemented using the codec's decoder method and are included in the list entries if `keepends` is true.

`sizehint`, if given, is passed as the `size` argument to the stream's `read()` method.

**reset()**
Resets the codec buffers used for keeping state.

Note that no stream repositioning should take place. This method is primarily intended to be able to recover from decoding errors.

In addition to the above methods, the `StreamReader` must also inherit all other methods and attributes from the underlying stream.

The next two base classes are included for convenience. They are not needed by the codec registry, but may provide useful in practice.

### 7.8.1.6. StreamReaderWriter Objects

The `StreamReaderWriter` allows wrapping streams which work in both read and write modes.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

```python
class codecs.StreamReaderWriter(stream, Reader, Writer, errors)
    Creates a StreamReaderWriter instance. stream must be a file-like
```
object. Reader and Writer must be factory functions or classes providing the StreamReader and StreamWriter interface resp. Error handling is done in the same way as defined for the stream readers and writers.

StreamReaderWriter instances define the combined interfaces of StreamReader and StreamWriter classes. They inherit all other methods and attributes from the underlying stream.

### 7.8.1.7. StreamRecoder Objects

The StreamRecoder provide a frontend - backend view of encoding data which is sometimes useful when dealing with different encoding environments.

The design is such that one can use the factory functions returned by the lookup() function to construct the instance.

```python
class codecs.StreamRecoder(stream, encode, decode, Reader, Writer, errors)
```

Creates a StreamRecoder instance which implements a two-way conversion: encode and decode work on the frontend (the input to read() and output of write()) while Reader and Writer work on the backend (reading and writing to the stream).

You can use these objects to do transparent direct recodings from e.g. Latin-1 to UTF-8 and back.

stream must be a file-like object.

encode, decode must adhere to the codec interface. Reader, Writer must be factory functions or classes providing objects of the StreamReader and StreamWriter interface respectively.

encode and decode are needed for the frontend translation,
Reader and Writer for the backend translation. The intermediate format used is determined by the two sets of codecs, e.g. the Unicode codecs will use Unicode as the intermediate encoding.

Error handling is done in the same way as defined for the stream readers and writers.

StreamRecoder instances define the combined interfaces of StreamReader and StreamWriter classes. They inherit all other methods and attributes from the underlying stream.
7.8.2. Encodings and Unicode

Unicode strings are stored internally as sequences of code points (to be precise as `Py_UNICODE` arrays). Depending on the way Python is compiled (either via `--enable-unicode=ucs2` or `--enable-unicode=ucs4`, with the former being the default) `Py_UNICODE` is either a 16-bit or 32-bit data type. Once a Unicode object is used outside of CPU and memory, CPU endianness and how these arrays are stored as bytes become an issue. Transforming a unicode object into a sequence of bytes is called encoding and recreating the unicode object from the sequence of bytes is known as decoding. There are many different methods for how this transformation can be done (these methods are also called encodings). The simplest method is to map the code points 0–255 to the bytes `0x0–0xff`. This means that a unicode object that contains code points above `U+00FF` can't be encoded with this method (which is called 'latin-1' or 'iso-8859-1'). `unicode.encode()` will raise a `UnicodeEncodeError` that looks like this: `UnicodeEncodeError: 'latin-1' codec can't encode character u'\u1234' in position 3: ordinal not in range(256).

There's another group of encodings (the so called charmap encodings) that choose a different subset of all unicode code points and how these code points are mapped to the bytes `0x0–0xff`. To see how this is done simply open e.g. `encodings/cp1252.py` (which is an encoding that is used primarily on Windows). There's a string constant with 256 characters that shows you which character is mapped to which byte value.

All of these encodings can only encode 256 of the 1114112 code points defined in unicode. A simple and straightforward way that can store each Unicode code point, is to store each code point as four consecutive bytes. There are two possibilities: store the bytes in big
 endian or in little endian order. These two encodings are called **UTF-32-BE** and **UTF-32-LE** respectively. Their disadvantage is that if e.g. you use **UTF-32-BE** on a little endian machine you will always have to swap bytes on encoding and decoding. **UTF-32** avoids this problem: bytes will always be in natural endianness. When these bytes are read by a CPU with a different endianness, then bytes have to be swapped though. To be able to detect the endianness of a **UTF-16** or **UTF-32** byte sequence, there’s the so called BOM (“Byte Order Mark”). This is the Unicode character **U+FEFF**. This character can be prepended to every **UTF-16** or **UTF-32** byte sequence. The byte swapped version of this character (0xFFFE) is an illegal character that may not appear in a Unicode text. So when the first character in an **UTF-16** or **UTF-32** byte sequence appears to be a **U+FFEFE** the bytes have to be swapped on decoding. Unfortunately the character **U+FEFF** had a second purpose as a **ZERO WIDTH NO-BREAK SPACE**: a character that has no width and doesn’t allow a word to be split. It can e.g. be used to give hints to a ligature algorithm. With Unicode 4.0 using **U+FEFF** as a **ZERO WIDTH NO-BREAK SPACE** has been deprecated (with **U+2060** (**WORD JOINER**) assuming this role). Nevertheless Unicode software still must be able to handle **U+FEFF** in both roles: as a BOM it’s a device to determine the storage layout of the encoded bytes, and vanishes once the byte sequence has been decoded into a Unicode string; as a **ZERO WIDTH NO-BREAK SPACE** it’s a normal character that will be decoded like any other.

There’s another encoding that is able to encoding the full range of Unicode characters: UTF-8. UTF-8 is an 8-bit encoding, which means there are no issues with byte order in UTF-8. Each byte in a UTF-8 byte sequence consists of two parts: marker bits (the most significant bits) and payload bits. The marker bits are a sequence of zero to four 1 bits followed by a 0 bit. Unicode characters are encoded like this (with x being payload bits, which when concatenated give the Unicode character):
<table>
<thead>
<tr>
<th>Range</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>U-000000000 ... U-00000007F</td>
<td>0xxxxxxx</td>
</tr>
<tr>
<td>U-000000080 ... U-000007FF</td>
<td>110xxxxx 10xxxxxx</td>
</tr>
<tr>
<td>U-000000800 ... U-00000FFFF</td>
<td>1110xxxx 10xxxxxx 10xxxxxx</td>
</tr>
<tr>
<td>U-00010000 ... U-0010FFFF</td>
<td>11110xxx 10xxxxxx 10xxxxxx 10xxxxxx</td>
</tr>
</tbody>
</table>

The least significant bit of the Unicode character is the rightmost x bit.

As UTF-8 is an 8-bit encoding no BOM is required and any U+FEFF character in the decoded Unicode string (even if it's the first character) is treated as a ZERO WIDTH NO-BREAK SPACE.

Without external information it's impossible to reliably determine which encoding was used for encoding a Unicode string. Each charmap encoding can decode any random byte sequence. However that's not possible with UTF-8, as UTF-8 byte sequences have a structure that doesn't allow arbitrary byte sequences. To increase the reliability with which a UTF-8 encoding can be detected, Microsoft invented a variant of UTF-8 (that Python 2.5 calls "utf-8-sig") for its Notepad program: Before any of the Unicode characters is written to the file, a UTF-8 encoded BOM (which looks like this as a byte sequence: 0xef, 0xbb, 0xbf) is written. As it's rather improbable that any charmap encoded file starts with these byte values (which would e.g. map to

LATIN SMALL LETTER I WITH DIAERESIS
RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
INVERTED QUESTION MARK

in iso-8859-1), this increases the probability that a utf-8-sig encoding can be correctly guessed from the byte sequence. So here the BOM is not used to be able to determine the byte order used for
generating the byte sequence, but as a signature that helps in guessing the encoding. On encoding the utf-8-sig codec will write \texttt{\textbackslash 0xef, \textbackslash 0xbb, \textbackslash 0xbf} as the first three bytes to the file. On decoding \texttt{utf-8-sig} will skip those three bytes if they appear as the first three bytes in the file. In UTF-8, the use of the BOM is discouraged and should generally be avoided.
Python comes with a number of codecs built-in, either implemented as C functions or with dictionaries as mapping tables. The following table lists the codecs by name, together with a few common aliases, and the languages for which the encoding is likely used. Neither the list of aliases nor the list of languages is meant to be exhaustive. Notice that spelling alternatives that only differ in case or use a hyphen instead of an underscore are also valid aliases; therefore, e.g. `utf-8` is a valid alias for the `utf_8` codec.

Many of the character sets support the same languages. They vary in individual characters (e.g. whether the EURO SIGN is supported or not), and in the assignment of characters to code positions. For the European languages in particular, the following variants typically exist:

- an ISO 8859 codeset
- a Microsoft Windows code page, which is typically derived from an 8859 codeset, but replaces control characters with additional graphic characters
- an IBM EBCDIC code page
- an IBM PC code page, which is ASCII compatible

<table>
<thead>
<tr>
<th>Codec</th>
<th>Aliases</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii</td>
<td>646, us-ascii</td>
<td>English</td>
</tr>
<tr>
<td>big5</td>
<td>big5-tw, csbig5</td>
<td>Traditional Chinese</td>
</tr>
<tr>
<td>big5hkscs</td>
<td>big5-hkscs, hkscs</td>
<td>Traditional Chinese</td>
</tr>
<tr>
<td>cp037</td>
<td>IBM037, IBM039</td>
<td>English</td>
</tr>
<tr>
<td>cp424</td>
<td>EBCDIC-CP-HE, IBM424</td>
<td>Hebrew</td>
</tr>
<tr>
<td>cp437</td>
<td>437, IBM437</td>
<td>English</td>
</tr>
<tr>
<td></td>
<td>EBCDIC-CP-BE,</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Region</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>cp500</td>
<td>EBCDIC-CP-CH, IBM500</td>
<td>Western Europe</td>
</tr>
<tr>
<td>cp720</td>
<td></td>
<td>Arabic</td>
</tr>
<tr>
<td>cp737</td>
<td></td>
<td>Greek</td>
</tr>
<tr>
<td>cp775</td>
<td>IBM775</td>
<td>Baltic languages</td>
</tr>
<tr>
<td>cp850</td>
<td>850, IBM850</td>
<td>Western Europe</td>
</tr>
<tr>
<td>cp852</td>
<td>852, IBM852</td>
<td>Central and Eastern Europe</td>
</tr>
<tr>
<td>cp855</td>
<td>855, IBM855</td>
<td>Bulgarian, Byelorussian, Macedonian, Russian, Serbian</td>
</tr>
<tr>
<td>cp856</td>
<td></td>
<td>Hebrew</td>
</tr>
<tr>
<td>cp857</td>
<td>857, IBM857</td>
<td>Turkish</td>
</tr>
<tr>
<td>cp858</td>
<td>858, IBM858</td>
<td>Western Europe</td>
</tr>
<tr>
<td>cp860</td>
<td>860, IBM860</td>
<td>Portuguese</td>
</tr>
<tr>
<td>cp861</td>
<td>861, CP-IS, IBM861</td>
<td>Icelandic</td>
</tr>
<tr>
<td>cp862</td>
<td>862, IBM862</td>
<td>Hebrew</td>
</tr>
<tr>
<td>cp863</td>
<td>863, IBM863</td>
<td>Canadian</td>
</tr>
<tr>
<td>cp864</td>
<td>IBM864</td>
<td>Arabic</td>
</tr>
<tr>
<td>cp865</td>
<td>865, IBM865</td>
<td>Danish, Norwegian</td>
</tr>
<tr>
<td>cp866</td>
<td>866, IBM866</td>
<td>Russian</td>
</tr>
<tr>
<td>cp869</td>
<td>869, CP-GR, IBM869</td>
<td>Greek</td>
</tr>
<tr>
<td>cp874</td>
<td></td>
<td>Thai</td>
</tr>
<tr>
<td>cp875</td>
<td></td>
<td>Greek</td>
</tr>
<tr>
<td>cp932</td>
<td>932, ms932, mskanji, ms-kanji</td>
<td>Japanese</td>
</tr>
<tr>
<td>cp949</td>
<td>949, ms949, uhc</td>
<td>Korean</td>
</tr>
<tr>
<td>cp950</td>
<td>950, ms950</td>
<td>Traditional Chinese</td>
</tr>
<tr>
<td>cp1006</td>
<td></td>
<td>Urdu</td>
</tr>
<tr>
<td>cp1026</td>
<td>ibm1026</td>
<td>Turkish</td>
</tr>
<tr>
<td>cp1140</td>
<td>ibm1140</td>
<td>Western Europe</td>
</tr>
<tr>
<td>cp1250</td>
<td>windows-1250</td>
<td>Central and Eastern Europe</td>
</tr>
<tr>
<td>Code</td>
<td>Windows Code</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>cp1251</td>
<td>windows-1251</td>
<td>Europe</td>
</tr>
<tr>
<td>cp1252</td>
<td>windows-1252</td>
<td>Western Europe</td>
</tr>
<tr>
<td>cp1253</td>
<td>windows-1253</td>
<td>Greek</td>
</tr>
<tr>
<td>cp1254</td>
<td>windows-1254</td>
<td>Turkish</td>
</tr>
<tr>
<td>cp1255</td>
<td>windows-1255</td>
<td>Hebrew</td>
</tr>
<tr>
<td>cp1256</td>
<td>windows-1256</td>
<td>Arabic</td>
</tr>
<tr>
<td>cp1257</td>
<td>windows-1257</td>
<td>Baltic languages</td>
</tr>
<tr>
<td>cp1258</td>
<td>windows-1258</td>
<td>Vietnamese</td>
</tr>
<tr>
<td>euc_jp</td>
<td>eucjp, ujis, u-jis</td>
<td>Japanese</td>
</tr>
<tr>
<td>euc_jis_2004</td>
<td>jisx0213, ucejis2004</td>
<td>Japanese</td>
</tr>
<tr>
<td>euc_jisx0213</td>
<td>eucjisx0213</td>
<td>Japanese</td>
</tr>
<tr>
<td>euc_kr</td>
<td>euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001</td>
<td>Korean</td>
</tr>
<tr>
<td>gb2312</td>
<td>chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58</td>
<td>Simplified Chinese</td>
</tr>
<tr>
<td>gbk</td>
<td>936, cp936, ms936</td>
<td>Unified Chinese</td>
</tr>
<tr>
<td>gb18030</td>
<td>gb18030-2000</td>
<td>Unified Chinese</td>
</tr>
<tr>
<td>hz</td>
<td>hzgb, hz-gb, hz-gb-2312</td>
<td>Simplified Chinese</td>
</tr>
<tr>
<td>iso2022.jp</td>
<td>csiso2022jp, iso2022jp, iso-2022-jp</td>
<td>Japanese</td>
</tr>
<tr>
<td>iso2022.jp_1</td>
<td>iso2022jp-1, iso-2022-jp-1</td>
<td>Japanese</td>
</tr>
<tr>
<td>iso2022.jp_2</td>
<td>iso2022jp-2, iso-2022-jp-2</td>
<td>Japanese, Korean, Simplified Chinese, Western Europe, Greek</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Language</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>iso2022_jp_3</td>
<td>iso2022jp-3, iso-2022-jp-3</td>
<td>Japanese</td>
</tr>
<tr>
<td>iso2022_jp_ext</td>
<td>iso2022jp-ext, iso-2022-jp-ext</td>
<td>Japanese</td>
</tr>
<tr>
<td>latin_1</td>
<td>iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1</td>
<td>West Europe</td>
</tr>
<tr>
<td>iso8859_2</td>
<td>iso-8859-2, latin2, L2</td>
<td>Central and Eastern Europe</td>
</tr>
<tr>
<td>iso8859_3</td>
<td>iso-8859-3, latin3, L3</td>
<td>Esperanto, Maltese</td>
</tr>
<tr>
<td>iso8859_4</td>
<td>iso-8859-4, latin4, L4</td>
<td>Baltic languages</td>
</tr>
<tr>
<td>iso8859_5</td>
<td>iso-8859-5, cyrillic</td>
<td>Bulgarian, Byelorussian, Macedonian, Russian, Serbian</td>
</tr>
<tr>
<td>iso8859_6</td>
<td>iso-8859-6, arabic</td>
<td>Arabic</td>
</tr>
<tr>
<td>iso8859_7</td>
<td>iso-8859-7, greek, greek8</td>
<td>Greek</td>
</tr>
<tr>
<td>iso8859_8</td>
<td>iso-8859-8, hebrew</td>
<td>Hebrew</td>
</tr>
<tr>
<td>iso8859_9</td>
<td>iso-8859-9, latin5, L5</td>
<td>Turkish</td>
</tr>
<tr>
<td>iso8859_10</td>
<td>iso-8859-10, latin6, L6</td>
<td>Nordic languages</td>
</tr>
<tr>
<td>iso8859_11</td>
<td>iso-8859-11, thai</td>
<td>Thai languages</td>
</tr>
<tr>
<td>iso8859_13</td>
<td>iso-8859-13, latin7, L7</td>
<td>Baltic languages</td>
</tr>
<tr>
<td>iso8859_14</td>
<td>iso-8859-14, latin8, L8</td>
<td>Celtic languages</td>
</tr>
<tr>
<td>iso8859_15</td>
<td>iso-8859-15, latin9, L9</td>
<td>Western Europe</td>
</tr>
<tr>
<td>iso8859_16</td>
<td>iso-8859-16, latin10, L10</td>
<td>South-Eastern Europe</td>
</tr>
<tr>
<td>johab</td>
<td>cp1361, ms1361</td>
<td>Korean</td>
</tr>
<tr>
<td>koi8_r</td>
<td></td>
<td>Russian</td>
</tr>
<tr>
<td>koi8_u</td>
<td></td>
<td>Ukrainian</td>
</tr>
<tr>
<td>Encoding</td>
<td>Description</td>
<td>Languages Supported</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>mac_cyrillic</td>
<td>maccyrillic</td>
<td>Bulgarian, Byelorussian, Macedonian, Russian, Serbian</td>
</tr>
<tr>
<td>mac_greek</td>
<td>macgreek</td>
<td>Greek</td>
</tr>
<tr>
<td>mac_iceland</td>
<td>maciceland</td>
<td>Icelandic</td>
</tr>
<tr>
<td>mac_latin2</td>
<td>maclatin2, maccentraleurope</td>
<td>Central and Eastern Europe</td>
</tr>
<tr>
<td>mac_roman</td>
<td>macroman</td>
<td>Western Europe</td>
</tr>
<tr>
<td>mac_turkish</td>
<td>macturkish</td>
<td>Turkish</td>
</tr>
<tr>
<td>ptcp154</td>
<td>csptcp154, pt154, cp154, cyrillic-asian</td>
<td>Kazakh</td>
</tr>
<tr>
<td>shift_jis</td>
<td>csshiftjis, shiftjis, sjis, s_jis</td>
<td>Japanese</td>
</tr>
<tr>
<td>shift_jisx0213</td>
<td>shiftjisx0213, sjisx0213, s_jisx0213</td>
<td>Japanese</td>
</tr>
<tr>
<td>utf_32</td>
<td>U32, utf32</td>
<td>all languages</td>
</tr>
<tr>
<td>utf_32_be</td>
<td>UTF-32BE</td>
<td>all languages</td>
</tr>
<tr>
<td>utf_32_le</td>
<td>UTF-32LE</td>
<td>all languages</td>
</tr>
<tr>
<td>utf_16</td>
<td>U16, utf16</td>
<td>all languages</td>
</tr>
<tr>
<td>utf_16_be</td>
<td>UTF-16BE</td>
<td>all languages (BMP only)</td>
</tr>
<tr>
<td>utf_16_le</td>
<td>UTF-16LE</td>
<td>all languages (BMP only)</td>
</tr>
<tr>
<td>utf_7</td>
<td>U7, unicode-1-1-utf-7</td>
<td>all languages</td>
</tr>
<tr>
<td>utf_8</td>
<td>U8, UTF, utf8</td>
<td>all languages</td>
</tr>
<tr>
<td>utf_8_sig</td>
<td></td>
<td>all languages</td>
</tr>
</tbody>
</table>
7.8.4. Python Specific Encodings

A number of predefined codecs are specific to Python, so their codec names have no meaning outside Python. These are listed in the tables below based on the expected input and output types (note that while text encodings are the most common use case for codecs, the underlying codec infrastructure supports arbitrary data transforms rather than just text encodings). For asymmetric codecs, the stated purpose describes the encoding direction.

The following codecs provide unicode-to-str encoding [1] and str-to-unicode decoding [2], similar to the Unicode text encodings.

<table>
<thead>
<tr>
<th>Codec</th>
<th>Aliases</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>idna</td>
<td></td>
<td>Implements <a href="https://tools.ietf.org/html/rfc3490">RFC 3490</a>, see also encodings.idna</td>
</tr>
<tr>
<td>mbcS</td>
<td>dbcs</td>
<td>Windows only: Encode operand according to the ANSI codepage (CP_ACP)</td>
</tr>
<tr>
<td>palmos</td>
<td></td>
<td>Encoding of PalmOS 3.5</td>
</tr>
<tr>
<td>punycode</td>
<td></td>
<td>Implements <a href="https://tools.ietf.org/html/rfc3492">RFC 3492</a></td>
</tr>
<tr>
<td>raw_unicode_escape</td>
<td></td>
<td>Produce a string that is suitable as raw Unicode literal in Python source code</td>
</tr>
<tr>
<td>rot_13</td>
<td>rot13</td>
<td>Returns the Caesar-cypher encryption of the operand</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Raise an exception for all conversions. Can be used as the</td>
</tr>
</tbody>
</table>
undefined  

system encoding if no automatic coercion between byte and Unicode strings is desired.

unicode_escape  

Produce a string that is suitable as Unicode literal in Python source code

unicode_internal  

Return the internal representation of the operand

---

**New in version 2.3:** The idna and punycode encodings.

The following codecs provide str-to-str encoding and decoding [2].

<table>
<thead>
<tr>
<th>Codec</th>
<th>Aliases</th>
<th>Purpose</th>
<th>Encoder/decoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64_codec</td>
<td>base64, base64</td>
<td>Convert operand to multiline MIME base64 (the result always includes a trailing \n)</td>
<td>base64.encodestring(), base64.decodestring()</td>
</tr>
<tr>
<td>bz2_codec</td>
<td>bz2</td>
<td>Compress the operand using bz2</td>
<td>bz2.compress(), bz2.decompress()</td>
</tr>
<tr>
<td>hex_codec</td>
<td>hex</td>
<td>Convert operand to hexadecimal representation, with two digits per byte</td>
<td>binascii.b2a_hex(), binascii.a2b_hex()</td>
</tr>
<tr>
<td>quopri_codec</td>
<td>quopri, quoted-printable, quotedprintable</td>
<td>Convert operand to MIME quoted</td>
<td>quopri.encode() with quotetabs=True, quopri.decode()</td>
</tr>
<tr>
<td>Package</td>
<td>Module</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>------------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>string_escape</td>
<td></td>
<td>Produce a string that is suitable as string literal in Python source code</td>
<td></td>
</tr>
<tr>
<td>uu_codec</td>
<td>uu</td>
<td>Convert the operand using uuencode</td>
<td>uu.encode(), uu.decode()</td>
</tr>
<tr>
<td>zlib_codec</td>
<td>zip, zlib</td>
<td>Compress the operand using gzip</td>
<td>zlib.compress(), zlib.decompress()</td>
</tr>
</tbody>
</table>

[1] str objects are also accepted as input in place of unicode objects. They are implicitly converted to unicode by decoding them using the default encoding. If this conversion fails, it may lead to encoding operations raising UnicodeDecodeError.

[2] unicode objects are also accepted as input in place of str objects. They are implicitly converted to str by encoding them using the default encoding. If this conversion fails, it may lead to decoding operations raising UnicodeEncodeError.
7.8.5. encodings.idna — Internationalized Domain Names in Applications

New in version 2.3.

This module implements RFC 3490 (Internationalized Domain Names in Applications) and RFC 3492 (Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)). It builds upon the punycode encoding and stringprep.

These RFCs together define a protocol to support non-ASCII characters in domain names. A domain name containing non-ASCII characters (such as www.Alliancefrançaise.nu) is converted into an ASCII-compatible encoding (ACE, such as www.xn--alliancefranaise-npb.nu). The ACE form of the domain name is then used in all places where arbitrary characters are not allowed by the protocol, such as DNS queries, HTTP Host fields, and so on. This conversion is carried out in the application; if possible invisible to the user: The application should transparently convert Unicode domain labels to IDNA on the wire, and convert back ACE labels to Unicode before presenting them to the user.

Python supports this conversion in several ways: the idna codec performs conversion between Unicode and ACE, separating an input string into labels based on the separator characters defined in section 3.1 (1) of RFC 3490 and converting each label to ACE as required, and conversely separating an input byte string into labels based on the . separator and converting any ACE labels found into unicode. Furthermore, the socket module transparently converts Unicode host names to ACE, so that applications need not be concerned about converting host names themselves when they pass them to the socket module. On top of that, modules that have host
names as function parameters, such as `httplib` and `ftplib`, accept Unicode host names (`httplib` then also transparently sends an IDNA hostname in the `Host` field if it sends that field at all).

When receiving host names from the wire (such as in reverse name lookup), no automatic conversion to Unicode is performed: Applications wishing to present such host names to the user should decode them to Unicode.

The module `encodings.idna` also implements the nameprep procedure, which performs certain normalizations on host names, to achieve case-insensitivity of international domain names, and to unify similar characters. The nameprep functions can be used directly if desired.

`encodings.idna.nameprep(label)`
Return the nameprepped version of `label`. The implementation currently assumes query strings, so `AllowUnassigned` is true.

`encodings.idna.ToASCII(label)`
Convert a label to ASCII, as specified in RFC 3490. `UseSTD3ASCIIIRules` is assumed to be false.

`encodings.idna.ToUnicode(label)`
Convert a label to Unicode, as specified in RFC 3490.
New in version 2.5.

This module implements a variant of the UTF-8 codec: On encoding a UTF-8 encoded BOM will be prepended to the UTF-8 encoded bytes. For the stateful encoder this is only done once (on the first write to the byte stream). For decoding an optional UTF-8 encoded BOM at the start of the data will be skipped.
7.9. unicodedata — Unicode Database

This module provides access to the Unicode Character Database which defines character properties for all Unicode characters. The data in this database is based on the UnicodeData.txt file version 5.2.0 which is publicly available from ftp://ftp.unicode.org/.

The module uses the same names and symbols as defined by the UnicodeData File Format 5.2.0 (see http://www.unicode.org/reports/tr44/tr44-4.html). It defines the following functions:

unicodedata. lookup(name)

    Look up character by name. If a character with the given name is found, return the corresponding Unicode character. If not found, KeyError is raised.

unicodedata. name(unichr[, default])

    Returns the name assigned to the Unicode character unichr as a string. If no name is defined, default is returned, or, if not given, ValueError is raised.

unicodedata. decimal(unichr[, default])

    Returns the decimal value assigned to the Unicode character unichr as integer. If no such value is defined, default is returned, or, if not given, ValueError is raised.

unicodedata. digit(unichr[, default])

    Returns the digit value assigned to the Unicode character unichr as integer. If no such value is defined, default is returned, or, if not given, ValueError is raised.
unicodedata\.numeric\((unichr[, default])\)
    Returns the numeric value assigned to the Unicode character
    \texttt{unichr} as float. If no such value is defined, \texttt{default} is returned, or, if
    not given, \texttt{ValueError} is raised.

unicodedata\.category\((unichr)\)
    Returns the general category assigned to the Unicode character
    \texttt{unichr} as string.

unicodedata\.bidirectional\((unichr)\)
    Returns the bidirectional class assigned to the Unicode character
    \texttt{unichr} as string. If no such value is defined, an empty string is
    returned.

unicodedata\.combining\((unichr)\)
    Returns the canonical combining class assigned to the Unicode character
    \texttt{unichr} as integer. Returns \(0\) if no combining class is
    defined.

unicodedata\.east_asian_width\((unichr)\)
    Returns the east asian width assigned to the Unicode character
    \texttt{unichr} as string.

    \textit{New in version 2.4.}

unicodedata\.mirrored\((unichr)\)
    Returns the mirrored property assigned to the Unicode character
    \texttt{unichr} as integer. Returns \(1\) if the character has been identified
    as a “mirrored” character in bidirectional text, \(0\) otherwise.

unicodedata\.decomposition\((unichr)\)
    Returns the character decomposition mapping assigned to the
    Unicode character \texttt{unichr} as string. An empty string is returned in
    case no such mapping is defined.
unicodedata.normalize(form, unistr)

Return the normal form form for the Unicode string unistr. Valid values for form are 'NFC', 'NFKC', 'NFD', and 'NFKD'.

The Unicode standard defines various normalization forms of a Unicode string, based on the definition of canonical equivalence and compatibility equivalence. In Unicode, several characters can be expressed in various ways. For example, the character U+00C7 (LATIN CAPITAL LETTER C WITH CEDILLA) can also be expressed as the sequence U+0043 (LATIN CAPITAL LETTER C) U+0327 (COMBINING CEDILLA).

For each character, there are two normal forms: normal form C and normal form D. Normal form D (NFD) is also known as canonical decomposition, and translates each character into its decomposed form. Normal form C (NFC) first applies a canonical decomposition, then composes pre-combined characters again.

In addition to these two forms, there are two additional normal forms based on compatibility equivalence. In Unicode, certain characters are supported which normally would be unified with other characters. For example, U+2160 (ROMAN NUMERAL ONE) is really the same thing as U+0049 (LATIN CAPITAL LETTER I). However, it is supported in Unicode for compatibility with existing character sets (e.g. gb2312).

The normal form KD (NFKD) will apply the compatibility decomposition, i.e. replace all compatibility characters with their equivalents. The normal form KC (NFKC) first applies the compatibility decomposition, followed by the canonical composition.

Even if two unicode strings are normalized and look the same to a human reader, if one has combining characters and the other doesn't, they may not compare equal.
New in version 2.3.

In addition, the module exposes the following constant:

```python
unicodedata.unidata_version
```

The version of the Unicode database used in this module.

New in version 2.3.

```python
unicodedata.ucd_3_2_0
```

This is an object that has the same methods as the entire module, but uses the Unicode database version 3.2 instead, for applications that require this specific version of the Unicode database (such as IDNA).

New in version 2.5.

Examples:

```python
>>> import unicodedata
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'u'
>>> unicodedata.name(u'/')
'SOLIDUS'
>>> unicodedata.decimal(u'9')
9
>>> unicodedata.decimal(u'a')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: not a decimal
>>> unicodedata.category(u'A')  # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional(u'\u0660')  # 'A'rabic, 'N'umber
'AN'
```
When identifying things (such as host names) in the internet, it is often necessary to compare such identifications for “equality”. Exactly how this comparison is executed may depend on the application domain, e.g. whether it should be case-insensitive or not. It may be also necessary to restrict the possible identifications, to allow only identifications consisting of “printable” characters.

**RFC 3454** defines a procedure for “preparing” Unicode strings in internet protocols. Before passing strings onto the wire, they are processed with the preparation procedure, after which they have a certain normalized form. The RFC defines a set of tables, which can be combined into profiles. Each profile must define which tables it uses, and what other optional parts of the stringprep procedure are part of the profile. One example of a stringprep profile is nameprep, which is used for internationalized domain names.

The module stringprep only exposes the tables from RFC 3454. As these tables would be very large to represent them as dictionaries or lists, the module uses the Unicode character database internally. The module source code itself was generated using the mkstringprep.py utility.

As a result, these tables are exposed as functions, not as data structures. There are two kinds of tables in the RFC: sets and mappings. For a set, stringprep provides the “characteristic function”, i.e. a function that returns true if the parameter is part of the set. For mappings, it provides the mapping function: given the key, it returns the associated value. Below is a list of all functions
available in the module.

\texttt{stringprep.in_table_a1}(code)

Determine whether \textit{code} is in tableA.1 (Unassigned code points in Unicode 3.2).

\texttt{stringprep.in_table_b1}(code)

Determine whether \textit{code} is in tableB.1 (Commonly mapped to nothing).

\texttt{stringprep.map_table_b2}(code)

Return the mapped value for \textit{code} according to tableB.2 (Mapping for case-folding used with NFKC).

\texttt{stringprep.map_table_b3}(code)

Return the mapped value for \textit{code} according to tableB.3 (Mapping for case-folding used with no normalization).

\texttt{stringprep.in_table_c11}(code)

Determine whether \textit{code} is in tableC.1.1 (ASCII space characters).

\texttt{stringprep.in_table_c12}(code)

Determine whether \textit{code} is in tableC.1.2 (Non-ASCII space characters).

\texttt{stringprep.in_table_c11_c12}(code)

Determine whether \textit{code} is in tableC.1 (Space characters, union of C.1.1 and C.1.2).

\texttt{stringprep.in_table_c21}(code)

Determine whether \textit{code} is in tableC.2.1 (ASCII control characters).

\texttt{stringprep.in_table_c22}(code)

Determine whether \textit{code} is in tableC.2.2 (Non-ASCII control
stringprep. \texttt{in\_table\_c21\_c22}(code)
    Determine whether \textit{code} is in table C.2 (Control characters, union
    of C.2.1 and C.2.2).

stringprep. \texttt{in\_table\_c3}(code)
    Determine whether \textit{code} is in table C.3 (Private use).

stringprep. \texttt{in\_table\_c4}(code)
    Determine whether \textit{code} is in table C.4 (Non-character code
    points).

stringprep. \texttt{in\_table\_c5}(code)
    Determine whether \textit{code} is in table C.5 (Surrogate codes).

stringprep. \texttt{in\_table\_c6}(code)
    Determine whether \textit{code} is in table C.6 (Inappropriate for plain
    text).

stringprep. \texttt{in\_table\_c7}(code)
    Determine whether \textit{code} is in table C.7 (Inappropriate for
    canonical representation).

stringprep. \texttt{in\_table\_c8}(code)
    Determine whether \textit{code} is in table C.8 (Change display
    properties or are deprecated).

stringprep. \texttt{in\_table\_c9}(code)
    Determine whether \textit{code} is in table C.9 (Tagging characters).

stringprep. \texttt{in\_table\_d1}(code)
    Determine whether \textit{code} is in table D.1 (Characters with
    bidirectional property “R” or “AL”).

stringprep. \texttt{in\_table\_d2}(code)
Determine whether code is in tableD.2 (Characters with bidirectional property “L”).
### 7.11. fpformat — Floating point conversions

*Deprecated since version 2.6:* The `fpformat` module has been removed in Python 3.

The `fpformat` module defines functions for dealing with floating point numbers representations in 100% pure Python.

**Note:** This module is unnecessary: everything here can be done using the `%` string interpolation operator described in the String Formatting Operations section.

The `fpformat` module defines the following functions and an exception:

```python
fpformat.fix(x, digs)
```

Format `x` as `[-]ddd.ddd` with `digs` digits after the point and at least one digit before. If `digs <= 0`, the decimal point is suppressed.

`x` can be either a number or a string that looks like one. `digs` is an integer.

Return value is a string.

```python
fpformat.sci(x, digs)
```

Format `x` as `[-]d.dddE[+-]ddd` with `digs` digits after the point and exactly one digit before. If `digs <= 0`, one digit is kept and the point is suppressed.

`x` can be either a real number, or a string that looks like one. `digs`
is an integer.

Return value is a string.

**exception** `fpformat.NotANumber`

Exception raised when a string passed to `fix()` or `sci()` as the `x` parameter does not look like a number. This is a subclass of `ValueError` when the standard exceptions are strings. The exception value is the improperly formatted string that caused the exception to be raised.

Example:

```python
>>> import fpformat
>>> fpformat.fix(1.23, 1)
'1.2'
```
8. Data Types

The modules described in this chapter provide a variety of specialized data types such as dates and times, fixed-type arrays, heap queues, synchronized queues, and sets.

Python also provides some built-in data types, in particular, `dict`, `list`, `set` (which along with `frozenset`, replaces the deprecated `sets` module), and `tuple`. The `str` class can be used to handle binary data and 8-bit text, and the `unicode` class to handle Unicode text.

The following modules are documented in this chapter:

- **8.1. datetime — Basic date and time types**
  - 8.1.1. Available Types
  - 8.1.2. `timedelta` Objects
  - 8.1.3. `date` Objects
  - 8.1.4. `datetime` Objects
  - 8.1.5. `time` Objects
  - 8.1.6. `tzinfo` Objects
  - 8.1.7. `strftime()` and `strptime()` Behavior
- **8.2. calendar — General calendar-related functions**
- **8.3. collections — High-performance container datatypes**
  - 8.3.1. `Counter` objects
  - 8.3.2. `deque` objects
    - 8.3.2.1. `deque` Recipes
  - 8.3.3. `defaultdict` objects
    - 8.3.3.1. `defaultdict` Examples
  - 8.3.4. `namedtuple()` Factory Function for Tuples with Named Fields
  - 8.3.5. `OrderedDict` objects
    - 8.3.5.1. `OrderedDict` Examples and Recipes
8.3.6. Collections Abstract Base Classes

- 8.4. heapq — Heap queue algorithm
  - 8.4.1. Basic Examples
  - 8.4.2. Priority Queue Implementation Notes
  - 8.4.3. Theory
- 8.5. bisect — Array bisection algorithm
  - 8.5.1. Searching Sorted Lists
  - 8.5.2. Other Examples
- 8.6. array — Efficient arrays of numeric values
- 8.7. sets — Unordered collections of unique elements
  - 8.7.1. Set Objects
  - 8.7.2. Example
  - 8.7.3. Protocol for automatic conversion to immutable
  - 8.7.4. Comparison to the built-in set types
- 8.8. sched — Event scheduler
  - 8.8.1. Scheduler Objects
- 8.9. mutex — Mutual exclusion support
  - 8.9.1. Mutex Objects
- 8.10. Queue — A synchronized queue class
  - 8.10.1. Queue Objects
- 8.11. weakref — Weak references
  - 8.11.1. Weak Reference Objects
  - 8.11.2. Example
- 8.12. UserDict — Class wrapper for dictionary objects
- 8.13. UserList — Class wrapper for list objects
- 8.15. types — Names for built-in types
- 8.16. new — Creation of runtime internal objects
- 8.17. copy — Shallow and deep copy operations
- 8.18. pprint — Data pretty printer
  - 8.18.1. PrettyPrinter Objects
  - 8.18.2. pprint Example
- 8.19. `repr` — Alternate `repr()` implementation
  - 8.19.1. Repr Objects
  - 8.19.2. Subclassing Repr Objects
8.1. datetime — Basic date and time types

New in version 2.3.

The datetime module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation. For related functionality, see also the time and calendar modules.

There are two kinds of date and time objects: “naive” and “aware”.

An aware object has sufficient knowledge of applicable algorithmic and political time adjustments, such as time zone and daylight saving time information, to locate itself relative to other aware objects. An aware object is used to represent a specific moment in time that is not open to interpretation [1].

A naive object does not contain enough information to unambiguously locate itself relative to other date/time objects. Whether a naive object represents Coordinated Universal Time (UTC), local time, or time in some other timezone is purely up to the program, just like it’s up to the program whether a particular number represents metres, miles, or mass. Naive objects are easy to understand and to work with, at the cost of ignoring some aspects of reality.

For applications requiring aware objects, datetime and time objects have an optional time zone information attribute, tzinfo, that can be set to an instance of a subclass of the abstract tzinfo class. These tzinfo objects capture information about the offset from UTC time, the time zone name, and whether Daylight Saving Time is in effect.
Note that no concrete `tzinfo` classes are supplied by the `datetime` module. Supporting timezones at whatever level of detail is required is up to the application. The rules for time adjustment across the world are more political than rational, and there is no standard suitable for every application.

The `datetime` module exports the following constants:

```python
datetime.MINYEAR
    The smallest year number allowed in a `date` or `datetime` object. `MINYEAR` is 1.
```

```python
datetime.MAXYEAR
    The largest year number allowed in a `date` or `datetime` object. `MAXYEAR` is 9999.
```

See also:

- **Module `calendar`**
  General calendar related functions.

- **Module `time`**
  Time access and conversions.
8.1.1. Available Types

class datetime.date
An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: year, month, and day.

class datetime.time
An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds (there is no notion of “leap seconds” here). Attributes: hour, minute, second, microsecond, and tzinfo.

class datetime.datetime
A combination of a date and a time. Attributes: year, month, day, hour, minute, second, microsecond, and tzinfo.

class datetime.timedelta
A duration expressing the difference between two date, time, or datetime instances to microsecond resolution.

class datetime.tzinfo
An abstract base class for time zone information objects. These are used by the datetime and time classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).

Objects of these types are immutable.

Objects of the date type are always naive.

An object of type time or datetime may be naive or aware. A datetime object d is aware if d.tzinfo is not None and
d.tzinfo.utcoffset(d) does not return None. If d.tzinfo is None, or if d.tzinfo is not None but d.tzinfo.utcoffset(d) returns None, d is naive. A time object t is aware if t.tzinfo is not None and t.tzinfo.utcoffset(None) does not return None. Otherwise, t is naive.

The distinction between naive and aware doesn't apply to timedelta objects.

Subclass relationships:

```
object
    timedelta
tzinfo
time
date
datetime
```
8.1.2. `timedelta` Objects

A `timedelta` object represents a duration, the difference between two dates or times.

```python
class datetime.timedelta([days[, seconds[, microseconds[, milliseconds[, minutes[, hours[, weeks]]]]]])
```

All arguments are optional and default to 0. Arguments may be ints, longs, or floats, and may be positive or negative.

Only `days`, `seconds` and `microseconds` are stored internally. Arguments are converted to those units:

- A millisecond is converted to 1000 microseconds.
- A minute is converted to 60 seconds.
- An hour is converted to 3600 seconds.
- A week is converted to 7 days.

and days, seconds and microseconds are then normalized so that the representation is unique, with

- $0 \leq \text{microseconds} < 1000000$
- $0 \leq \text{seconds} < 3600 \times 24$ (the number of seconds in one day)
- $-999999999 \leq \text{days} \leq 999999999$

If any argument is a float and there are fractional microseconds, the fractional microseconds left over from all arguments are combined and their sum is rounded to the nearest microsecond. If no argument is a float, the conversion and normalization processes are exact (no information is lost).

If the normalized value of days lies outside the indicated range, `OverflowError` is raised.
Note that normalization of negative values may be surprising at first. For example,

```python
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

Class attributes are:

`timedelta.min`

The most negative `timedelta` object, `timedelta(-999999999)`.

`timedelta.max`

The most positive `timedelta` object, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

`timedelta.resolution`

The smallest possible difference between non-equal `timedelta` objects, `timedelta(microseconds=1)`.

Note that, because of normalization, `timedelta.max > -timedelta.min`. `-timedelta.max` is not representable as a `timedelta` object.

Instance attributes (read-only):

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>days</td>
<td>Between -999999999 and 999999999 inclusive</td>
</tr>
<tr>
<td>seconds</td>
<td>Between 0 and 86399 inclusive</td>
</tr>
<tr>
<td>microseconds</td>
<td>Between 0 and 999999 inclusive</td>
</tr>
</tbody>
</table>

Supported operations:

<p>| Operation | Result |</p>
<table>
<thead>
<tr>
<th>expression</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>t1 = t2 + t3</code></td>
<td>Sum of <code>t2</code> and <code>t3</code>. Afterwards <code>t1-t2 == t3</code> and <code>t1-t3 == t2</code> are true. (1)</td>
</tr>
<tr>
<td><code>t1 = t2 - t3</code></td>
<td>Difference of <code>t2</code> and <code>t3</code>. Afterwards <code>t1 == t2 - t3</code> and <code>t2 == t1 + t3</code> are true. (1)</td>
</tr>
<tr>
<td><code>t1 = t2 * i or t1 = i * t2</code></td>
<td>Delta multiplied by an integer or long. Afterwards <code>t1 // i == t2</code> is true, provided <code>i != 0</code>.</td>
</tr>
<tr>
<td><code>t1 = t2 // i</code></td>
<td>The floor is computed and the remainder (if any) is thrown away. (3)</td>
</tr>
<tr>
<td><code>+t1</code></td>
<td>Returns a <code>timedelta</code> object with the same value. (2)</td>
</tr>
<tr>
<td><code>-t1</code></td>
<td>equivalent to <code>timedelta(-t1.days, -t1.seconds, -t1.microseconds)</code>, and to <code>t1* -1</code>. (1)(4)</td>
</tr>
<tr>
<td><code>abs(t)</code></td>
<td>equivalent to <code>+t</code> when <code>t.days &gt;= 0</code>, and to <code>-t</code> when <code>t.days &lt; 0</code>. (2)</td>
</tr>
<tr>
<td><code>str(t)</code></td>
<td>Returns a string in the form <code>[D day[s], ][H]H:MM:SS.[uuuuuu]</code>, where <code>D</code> is negative for negative <code>t</code>. (5)</td>
</tr>
<tr>
<td><code>repr(t)</code></td>
<td>Returns a string in the form <code>datetime.timedelta(D[, S[, U]])</code>, where <code>D</code> is negative for negative <code>t</code>. (5)</td>
</tr>
</tbody>
</table>

Notes:

1. This is exact, but may overflow.
2. This is exact, and cannot overflow.
3. Division by 0 raises `ZeroDivisionError`.
4. `-timedelta.max` is not representable as a `timedelta` object.
5. String representations of `timedelta` objects are normalized
similarly to their internal representation. This leads to somewhat unusual results for negative timedeltas. For example:

```python
>>> timedelta(hours=-5)
datetime.timedelta(-1, 68400)
>>> print(_)
-1 day, 19:00:00
```

In addition to the operations listed above `timedelta` objects support certain additions and subtractions with `date` and `datetime` objects (see below).

Comparisons of `timedelta` objects are supported with the `timedelta` object representing the smaller duration considered to be the smaller timedelta. In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a `timedelta` object is compared to an object of a different type, `TypeError` is raised unless the comparison is `==` or `!=`. The latter cases return `False` or `True`, respectively.

`timedelta` objects are `hashable` (usable as dictionary keys), support efficient pickling, and in Boolean contexts, a `timedelta` object is considered to be true if and only if it isn’t equal to `timedelta(0)`.

Instance methods:

```python
timedelta.total_seconds()
```

Return the total number of seconds contained in the duration. Equivalent to `(td.microseconds + (td.seconds + td.days * 24 * 3600) * 10**6) / 10**6` computed with true division enabled.

Note that for very large time intervals (greater than 270 years on most platforms) this method will lose microsecond accuracy.

`New in version 2.7.`
Example usage:

```python
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23, minutes=50, seconds=600)  # adds up to 365 days
>>> year.total_seconds()
31536000.0
>>> year == another_year
True
>>> ten_years = 10 * year
>>> ten_years, ten_years.days // 365
(datetime.timedelta(3650), 10)
>>> nine_years = ten_years - year
>>> nine_years, nine_years.days // 365
(datetime.timedelta(3285), 9)
>>> three_years = nine_years // 3;
>>> three_years, three_years.days // 365
(datetime.timedelta(1095), 3)
>>> abs(three_years - ten_years) == 2 * three_years + year
True
```
8.1.3. date Objects

A `date` object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar indefinitely extended in both directions. January 1 of year 1 is called day number 1, January 2 of year 1 is called day number 2, and so on. This matches the definition of the “proleptic Gregorian” calendar in Dershowitz and Reingold's book Calendrical Calculations, where it's the base calendar for all computations. See the book for algorithms for converting between proleptic Gregorian ordinals and many other calendar systems.

```python
class datetime.date(year, month, day)
```

All arguments are required. Arguments may be ints or longs, in the following ranges:

- \( \text{MINYEAR} \leq \text{year} \leq \text{MAXYEAR} \)
- \( 1 \leq \text{month} \leq 12 \)
- \( 1 \leq \text{day} \leq \text{number of days in the given month and year} \)

If an argument outside those ranges is given, \text{ValueError} is raised.

Other constructors, all class methods:

```python
classmethod date.today()
```

Return the current local date. This is equivalent to

```python
date.fromtimestamp(time.time())
```

```python
classmethod date.fromtimestamp(timestamp)
```

Return the local date corresponding to the POSIX timestamp, such as is returned by \text{time.time()}. This may raise \text{ValueError}, if the timestamp is out of the range of values supported by the platform C \text{localtime()} function. It's common for this to be
restricted to years from 1970 through 2038. Note that on non-
POSIX systems that include leap seconds in their notion of a
timestamp, leap seconds are ignored by `fromtimestamp()`.

**classmethod date.strptime(ordinal)**

Return the date corresponding to the proleptic Gregorian ordinal,
where January 1 of year 1 has ordinal 1. `ValueError` is raised
unless `1 <= ordinal <= date.max.toordinal()`. For any date `d`,
`date.strptime(d.toordinal()) == d`.

**Class attributes:**

`date.min`

The earliest representable date, `date(MINYEAR, 1, 1)`.

`date.max`

The latest representable date, `date(MAXYEAR, 12, 31)`.

`date.resolution`

The smallest possible difference between non-equal date objects,
`timedelta(days=1)`.

**Instance attributes (read-only):**

`date.year`

Between `MINYEAR` and `MAXYEAR` inclusive.

`date.month`

Between 1 and 12 inclusive.

`date.day`

Between 1 and the number of days in the given month of the
given year.

**Supported operations:**
<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>date2 = date1 + timedelta</code></td>
<td><code>date2</code> is <code>timedelta.days</code> days removed from <code>date1</code>. (1)</td>
</tr>
<tr>
<td><code>date2 = date1 - timedelta</code></td>
<td>Computes <code>date2</code> such that <code>date2 + timedelta == date1</code>. (2)</td>
</tr>
<tr>
<td><code>timedelta = date1 - date2</code></td>
<td>(3)</td>
</tr>
<tr>
<td><code>date1 &lt; date2</code></td>
<td><code>date1</code> is considered less than <code>date2</code> when <code>date1</code> precedes <code>date2</code> in time. (4)</td>
</tr>
</tbody>
</table>

Notes:

1. `date2` is moved forward in time if `timedelta.days > 0`, or backward if `timedelta.days < 0`. Afterward `date2 - date1 == timedelta.days * timedelta.seconds + timedelta.microseconds` are ignored. `OverflowError` is raised if `date2.year` would be smaller than `MINYEAR` or larger than `MAXYEAR`.
2. This isn’t quite equivalent to `date1 + (-timedelta)`, because `-timedelta` in isolation can overflow in cases where `date1 - timedelta` does not. `timedelta.seconds` and `timedelta.microseconds` are ignored.
3. This is exact, and cannot overflow. `timedelta.seconds` and `timedelta.microseconds` are 0, and `date2 + timedelta == date1` after.
4. In other words, `date1 < date2` if and only if `date1.toordinal() < date2.toordinal()`. In order to stop comparison from falling back to the default scheme of comparing object addresses, date comparison normally raises `TypeError` if the other comparand isn’t also a `date` object. However, `NotImplemented` is returned instead if the other comparand has a `timetuple()` attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a `date` object is compared to an object of a different type, `TypeError` is raised unless the
comparison is == or !=. The latter cases return False or True, respectively.

Dates can be used as dictionary keys. In Boolean contexts, all date objects are considered to be true.

Instance methods:

date. replace(year, month, day)

Return a date with the same value, except for those parameters given new values by whichever keyword arguments are specified. For example, if d == date(2002, 12, 31), then d.replace(day=26) == date(2002, 12, 26).

date. timetuple()

Return a time.struct_time such as returned by time.localtime(). The hours, minutes and seconds are 0, and the DST flag is -1. d.timetuple() is equivalent to time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1)), where yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1 is the day number within the current year starting with 1 for January 1st.

date. toordinal()

Return the proleptic Gregorian ordinal of the date, where January 1 of year 1 has ordinal 1. For any date object d, date.fromordinal(d.toordinal()) == d.

date. weekday()

Return the day of the week as an integer, where Monday is 0 and Sunday is 6. For example, date(2002, 12, 4).weekday() == 2, a Wednesday. See also isoweekday().

date. isoweekday()
Return the day of the week as an integer, where Monday is 1 and Sunday is 7. For example, `date(2002, 12, 4).isoweekday() == 3`, a Wednesday. See also `weekday()`, `isocalendar()`.

**date.isocalendar()**
Return a 3-tuple, (ISO year, ISO week number, ISO weekday).

The ISO calendar is a widely used variant of the Gregorian calendar. See https://www.staff.science.uu.nl/~gent0113/calendar/isocalendar.htm for a good explanation.

The ISO year consists of 52 or 53 full weeks, and where a week starts on a Monday and ends on a Sunday. The first week of an ISO year is the first (Gregorian) calendar week of a year containing a Thursday. This is called week number 1, and the ISO year of that Thursday is the same as its Gregorian year.

For example, 2004 begins on a Thursday, so the first week of ISO year 2004 begins on Monday, 29 Dec 2003 and ends on Sunday, 4 Jan 2004, so that `date(2003, 12, 29).isocalendar() == (2004, 1, 1)` and `date(2004, 1, 4).isocalendar() == (2004, 1, 7)`.

**date.isoformat()**
Return a string representing the date in ISO 8601 format, ‘YYYY-MM-DD’. For example, `date(2002, 12, 4).isoformat() == '2002-12-04'`.

**date.__str__()**
For a date `d`, `str(d)` is equivalent to `d.isoformat()`.

**date.ctime()**
Return a string representing the date, for example `date(2002, 12, 4).ctime()`.
d.ctime() is equivalent to \texttt{time.ctime(time.mktime(d.timetuple()))} on platforms where the native C \texttt{ctime()} function (which \texttt{time.ctime()} invokes, but which \texttt{date.ctime()} does not invoke) conforms to the C standard.

\texttt{date.\texttt{strftime}(format)}

Return a string representing the date, controlled by an explicit format string. Format codes referring to hours, minutes or seconds will see 0 values. For a complete list of formatting directives, see section \texttt{strftime()} and \texttt{strptime()} Behavior.

\texttt{date.__format__}(format)

Same as \texttt{date.strftime()}. This makes it possible to specify a format string for a \texttt{date} object when using \texttt{str.format()}. See section \texttt{strftime()} and \texttt{strptime()} Behavior.

Example of counting days to an event:

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

Example of working with \texttt{date}:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 000
```
```python
>>> d
datetime.date(2002, 3, 11)
>>> t = d.timetuple()
>>> for i in t:
...    print i
2002          # year
3            # month
11           # day
0
0
0
0  # weekday (0 = Monday)
70         # 70th day in the year
-1
>>> ic = d.isocalendar()
>>> for i in ic:
...    print i
2002          # ISO year
11           # ISO week number
1      # ISO day number ( 1 = Monday )

>>> d.isoformat()
'2002-03-11'
>>> d.strftime('%d/%m/%Y')
'11/03/02'
>>> d.strftime('%A %d. %B %Y')
'Monday 11. March 2002'

>>> 'The {1} is {0:%d}, the {2} is {0:%B}'
'The day is 11, the month is March.'
```
8.1.4. datetime Objects

A datetime object is a single object containing all the information from a date object and a time object. Like a date object, datetime assumes the current Gregorian calendar extended in both directions; like a time object, datetime assumes there are exactly 3600*24 seconds in every day.

Constructor:

class datetime.datetime(year, month, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]])

The year, month and day arguments are required. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments may be ints or longs, in the following ranges:

- MINYEAR <= year <= MAXYEAR
- 1 <= month <= 12
- 1 <= day <= number of days in the given month and year
- 0 <= hour < 24
- 0 <= minute < 60
- 0 <= second < 60
- 0 <= microsecond < 1000000

If an argument outside those ranges is given, ValueError is raised.

Other constructors, all class methods:

classmethod datetime.now()

Return the current local datetime, with tzinfo None. This is equivalent to datetime.fromtimestamp(time.time()). See also
now(), fromtimestamp().

**classmethod datetime.now([tz])**

Return the current local date and time. If optional argument tz is None or not specified, this is like today(), but, if possible, supplies more precision than can be gotten from going through a time.time() timestamp (for example, this may be possible on platforms supplying the C gettimeofday() function).

If tz is not None, it must be an instance of a tzinfo subclass, and the current date and time are converted to tz's time zone. In this case the result is equivalent to tz.fromutc(datetime.utcnow().replace(tzinfo=tz)). See also today(), utcnow().

**classmethod datetime.utcnow()**

Return the current UTC date and time, with tzinfo None. This is like now(), but returns the current UTC date and time, as a naive datetime object. See also now().

**classmethod datetime.fromtimestamp(timestamp[, tz])**

Return the local date and time corresponding to the POSIX timestamp, such as is returned by time.time(). If optional argument tz is None or not specified, the timestamp is converted to the platform's local date and time, and the returned datetime object is naive.

If tz is not None, it must be an instance of a tzinfo subclass, and the timestamp is converted to tz's time zone. In this case the result is equivalent to tz.fromutc(datetime.utctfromtimestamp(timestamp).replace(tzinfo=tzinfo):

fromtimestamp() may raise ValueError, if the timestamp is out of
the range of values supported by the platform C `localtime()` or `gmtime()` functions. It's common for this to be restricted to years in 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by `fromtimestamp()`, and then it's possible to have two timestamps differing by a second that yield identical `datetime` objects. See also `utcfromtimestamp()`.

classmethod `datetime.utcfromtimestamp(timestamp)`
Return the UTC `datetime` corresponding to the POSIX timestamp, with `tzinfo None`. This may raise `ValueError`, if the timestamp is out of the range of values supported by the platform C `gmtime()` function. It's common for this to be restricted to years in 1970 through 2038. See also `fromtimestamp()`.

classmethod `datetime.fromordinal(ordinal)`
Return the `datetime` corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. `ValueError` is raised unless `1 <= ordinal <= datetime.max.toordinal()`. The hour, minute, second and microsecond of the result are all 0, and `tzinfo` is `None`.

classmethod `datetime.combine(date, time)`
Return a new `datetime` object whose date components are equal to the given `date` object's, and whose time components and `tzinfo` attributes are equal to the given `time` object's. For any `datetime` object `d`, `d == datetime.combine(d.date(), d.timetz())`. If `date` is a `datetime` object, its time components and `tzinfo` attributes are ignored.

classmethod `datetime.strptime(date_string, format)`
Return a `datetime` corresponding to `date_string`, parsed
according to format. This is equivalent to \texttt{datetime(* (time.strptime(date\_string, format)[0:6]))}. \texttt{ValueError} is raised if the date\_string and format can't be parsed by \texttt{time.strptime()} or if it returns a value which isn't a time tuple. For a complete list of formatting directives, see section \texttt{strftime()} and \texttt{strptime()} Behavior.

\textit{New in version 2.5.}

Class attributes:

\texttt{datetime.min}

The earliest representable \texttt{datetime}, \texttt{datetime(MINYEAR, 1, 1, tzinfo=None)}.

\texttt{datetime.max}

The latest representable \texttt{datetime}, \texttt{datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)}.

\texttt{datetime.resolution}

The smallest possible difference between non-equal \texttt{datetime} objects, \texttt{timedelta(microseconds=1)}.

Instance attributes (read-only):

\texttt{datetime.year}

Between \texttt{MINYEAR} and \texttt{MAXYEAR} inclusive.

\texttt{datetime.month}

Between 1 and 12 inclusive.

\texttt{datetime.day}

Between 1 and the number of days in the given month of the given year.

\texttt{datetime.hour}
In range(24).

```
datetime.minute
```
In range(60).

```
datetime.second
```
In range(60).

```
datetime.microsecond
```
In range(1000000).

```
datetime.tzinfo
```
The object passed as the `tzinfo` argument to the `datetime` constructor, or `None` if none was passed.

Supported operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datetime2 = datetime1 + timedelta</code></td>
<td>(1)</td>
</tr>
<tr>
<td><code>datetime2 = datetime1 - timedelta</code></td>
<td>(2)</td>
</tr>
<tr>
<td><code>timedelta = datetime1 - datetime2</code></td>
<td>(3)</td>
</tr>
<tr>
<td><code>datetime1 &lt; datetime2</code></td>
<td>Compares <code>datetime</code> to <code>datetime</code>. (4)</td>
</tr>
</tbody>
</table>

1. `datetime2` is a duration of `timedelta` removed from `datetime1`, moving forward in time if `timedelta.days` > 0, or backward if `timedelta.days` < 0. The result has the same `tzinfo` attribute as the input `datetime`, and `datetime2 - datetime1 == timedelta` after. `OverflowError` is raised if `datetime2.year` would be smaller than `MINYEAR` or larger than `MAXYEAR`. Note that no time zone adjustments are done even if the input is an aware object.

2. Computes the `datetime2` such that `datetime2 + timedelta == datetime1`. As for addition, the result has the same `tzinfo` attribute as the input `datetime`, and no time zone adjustments
are done even if the input is aware. This isn’t quite equivalent to datetime1 + (-timedelta), because -timedelta in isolation can overflow in cases where datetime1 - timedelta does not.

3. Subtraction of a `datetime` from a `datetime` is defined only if both operands are naive, or if both are aware. If one is aware and the other is naive, `TypeError` is raised.

   If both are naive, or both are aware and have the same `tzinfo` attribute, the `tzinfo` attributes are ignored, and the result is a `timedelta` object `t` such that `datetime2 + t == datetime1`. No time zone adjustments are done in this case.

   If both are aware and have different `tzinfo` attributes, `a-b` acts as if `a` and `b` were first converted to naive UTC datetimes first. The result is `(a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())` except that the implementation never overflows.

4. `datetime1` is considered less than `datetime2` when `datetime1` precedes `datetime2` in time.

   If one operand is naive and the other is aware, `TypeError` is raised. If both operands are aware, and have the same `tzinfo` attribute, the common `tzinfo` attribute is ignored and the base datetimes are compared. If both operands are aware and have different `tzinfo` attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from `self.utcoffset()`).

**Note:** In order to stop comparison from falling back to the default scheme of comparing object addresses, `datetime` comparison normally raises `TypeError` if the other comparand isn’t also a `datetime` object. However, `NotImplemented` is
returned instead if the other comparand has a `timetuple()` attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a `datetime` object is compared to an object of a different type, `TypeError` is raised unless the comparison is `==` or `!=`. The latter cases return `False` or `True`, respectively.

`datetime` objects can be used as dictionary keys. In Boolean contexts, all `datetime` objects are considered to be true.

Instance methods:

`datetime.date()`
- Return `date` object with same year, month and day.

`datetime.time()`
- Return `time` object with same hour, minute, second and microsecond. `tzinfo` is `None`. See also method `timetz()`.

`datetime.timetz()`
- Return `time` object with same hour, minute, second, microsecond, and `tzinfo` attributes. See also method `time()`.

`datetime.replace([year[, month[, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]]]]])`
- Return a `datetime` with the same attributes, except for those attributes given new values by whichever keyword arguments are specified. Note that `tzinfo=None` can be specified to create a naive `datetime` from an aware `datetime` with no conversion of date and time data.

`datetime.astimezone(tz)`
- Return a `datetime` object with new `tzinfo` attribute `tz`, adjusting
the date and time data so the result is the same UTC time as `self`,
but in `tz`'s local time.

tz must be an instance of a `tzinfo` subclass, and its `utcoffset()`
and `dst()` methods must not return `None`. `self` must be aware
(`self.tzinfo` must not be `None`, and `self.utcoffset()` must not
return `None`).

If `self.tzinfo` is `tz`, `self.astimezone(tz)` is equal to `self`: no
adjustment of date or time data is performed. Else the result is
local time in time zone `tz`, representing the same UTC time as
`self`: after `astz = dt.astimezone(tz), astz - astz.utcoffset()`
will usually have the same date and time data as `dt - dt.utcoffset()`.
The discussion of class `tzinfo` explains the
cases at Daylight Saving Time transition boundaries where this
cannot be achieved (an issue only if `tz` models both standard and
daylight time).

If you merely want to attach a time zone object `tz` to a datetime `dt`
without adjustment of date and time data, use
`dt.replace(tzinfo=tz)`. If you merely want to remove the time
zone object from an aware datetime `dt` without conversion of date
and time data, use `dt.replace(tzinfo=None)`.

Note that the default `tzinfo.fromutc()` method can be overridden
in a `tzinfo` subclass to affect the result returned by `astimezone()`.
Ignoring error cases, `astimezone()` acts like:

```python
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
        # Convert self to UTC, and attach the new time zone obje
utc = (self - self.utcoffset()).replace(tzinfo=tz)
        # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```
**datetime.utcoffset()**
If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.utcoffset(self)`, and raises an exception if the latter doesn’t return `None`, or a `timedelta` object representing a whole number of minutes with magnitude less than one day.

**datetime.dst()**
If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.dst(self)`, and raises an exception if the latter doesn’t return `None`, or a `timedelta` object representing a whole number of minutes with magnitude less than one day.

**datetime.tzname()**
If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.tzname(self)`, raises an exception if the latter doesn’t return `None` or a string object,

**datetime.timetuple()**
Return a `time.struct_time` such as returned by `time.localtime()`. `d.timetuple()` is equivalent to `time.struct_time((d.year, d.month, d.day, d.hour, d.minute, d.second, d.weekday(), yday, dst))`, where `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` is the day number within the current year starting with 1 for January 1st. The `tm_isdst` flag of the result is set according to the `dst()` method: `tzinfo` is `None` or `dst()` returns `None`, `tm_isdst` is set to `-1`; else if `dst()` returns a non-zero value, `tm_isdst` is set to `1`; else `tm_isdst` is set to `0`.

**datetime.utctimetuple()**
If `datetime` instance `d` is naive, this is the same as `d.timetuple()` except that `tm_isdst` is forced to 0 regardless of what `d.dst()` returns. DST is never in effect for a UTC time.
If $d$ is aware, $d$ is normalized to UTC time, by subtracting $d$.utcoffset(), and a `time.struct_time` for the normalized time is returned. `tm_isdst` is forced to 0. Note that the result's `tm_year` member may be `MINYEAR-1` or `MAXYEAR+1`, if $d$.year was `MINYEAR` or `MAXYEAR` and UTC adjustment spills over a year boundary.

```python
datetime.toordinal()
Return the proleptic Gregorian ordinal of the date. The same as `self.date().toordinal()`.
```

```python
datetime.weekday()
Return the day of the week as an integer, where Monday is 0 and Sunday is 6. The same as `self.date().weekday()`. See also `isoweekday()`.
```

```python
datetime.isoweekday()
Return the day of the week as an integer, where Monday is 1 and Sunday is 7. The same as `self.date().isoweekday()`. See also `weekday()`, `isocalendar()`.
```

```python
datetime.isocalendar()
Return a 3-tuple, (ISO year, ISO week number, ISO weekday). The same as `self.date().isocalendar()`.
```

```python
datetime.isoformat([sep])
Return a string representing the date and time in ISO 8601 format, YYYY-MM-DDTHH:MM:SS.mmmmmmm or, if `microsecond` is 0, YYYY-MM-DDTHH:MM:SS

If `utcoffset()` does not return `None`, a 6-character string is appended, giving the UTC offset in (signed) hours and minutes: YYYY-MM-DDTHH:MM:SS.mmmmmmm+HH:MM or, if `microsecond` is 0 YYYY-MM-DDTHH:MM:SS+HH:MM
```
The optional argument `sep` (default 'T') is a one-character separator, placed between the date and time portions of the result. For example,

```python
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(minutes=-3)
... >>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat('T')
'2002-12-25 00:00:00-06:39'
```

datetime.__str__()

For a `datetime` instance `d`, `str(d)` is equivalent to `d.isoformat('T')`.

datetime.ctime()

Return a string representing the date and time, for example `datetime(2002, 12, 4, 20, 30, 40).ctime() == 'Wed Dec 4 20:30:40 2002'`. `d.ctime()` is equivalent to `time.ctime(time.mktime(d.timetuple()))` on platforms where the native C `ctime()` function (which `time.ctime()` invokes, but which `datetime.ctime()` does not invoke) conforms to the C standard.

datetime.strftime(format)

Return a string representing the date and time, controlled by an explicit format string. For a complete list of formatting directives, see section `strftime()` and `strptime()` Behavior.

datetime.__format__(format)

Same as `datetime.strftime()`. This makes it possible to specify a format string for a `datetime` object when using `str.format()`. See section `strftime()` and `strptime()` Behavior.

Examples of working with `datetime` objects:
from datetime import datetime, date, time

# Using datetime.combine()

d = date(2005, 7, 14)
t = time(12, 30)
datetime.datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)

# Using datetime.now() or datetime.utcnow()

datetime.datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)  # GMT +1
datetime.datetime.utcnow()
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060)

# Using datetime.strptime()

dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
dt
datetime.datetime(2006, 11, 21, 16, 30)

# Using datetime.timetuple() to get tuple of all attributes

tt = dt.timetuple()
for it in tt:
...
print it
...
2006  # year
11  # month
21  # day
16  # hour
30  # minute
0  # second
1  # weekday (0 = Monday)
325  # number of days since 1st January
-1  # dst - method tzinfo.dst() returned None

# Date in ISO format

ic = dt.isocalendar()

for it in ic:
...
print it
...
2006  # ISO year
47  # ISO week
2  # ISO weekday

# Formatting datetime

dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'

'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M}.
The day is 21, the month is November, the time is 04:30PM.'

Using datetime with tzinfo:
```python
>>> from datetime import timedelta, datetime, tzinfo

>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1) + self.dst(dt)
...     def dst(self, dt):
...         # DST starts last Sunday in March
...         d = datetime(dt.year, 4, 1)  # ends last Sunday in November
...         self.dston = d - timedelta(days=d.weekday() + 1)
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday() + 1)
...         if self.dston <= dt.replace(tzinfo=None) < self.dstoff:
...             return timedelta(hours=1)
...         else:
...             return timedelta(0)
...     def tzname(self, dt):
...         return "GMT +1"

>>> class GMT2(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=2) + self.dst(dt)
...     def dst(self, dt):
...         d = datetime(dt.year, 4, 1)
...         self.dston = d - timedelta(days=d.weekday() + 1)
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday() + 1)
...         if self.dston <= dt.replace(tzinfo=None) < self.dstoff:
...             return timedelta(hours=1)
...         else:
...             return timedelta(0)
...     def tzname(self, dt):
...         return "GMT +2"

>>> gmt1 = GMT1()
>>> # Daylight Saving Time
>>> dt1 = datetime(2006, 11, 21, 16, 30, tzinfo=gmt1)
>>> dt1.dst()
datetime.timedelta(0)
>>> dt1.utcoffset()
datetime.timedelta(3600)
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=gmt1)
>>> dt2.dst()
datetime.timedelta(0, 3600)
>>> dt2.utcoffset()
datetime.timedelta(0, 7200)
>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(GMT2())
>>> dt3
```

datetime.datetime(2006, 6, 14, 14, 0, tzinfo=<GMT2 object at 0x
>>> dt2
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=<GMT1 object at 0x
>>> dt2.utctimetuple() == dt3.utctimetuple()
True
8.1.5. time Objects

A time object represents a (local) time of day, independent of any particular day, and subject to adjustment via a tzinfo object.

```python
class datetime.time([hour[, minute[, second[, microsecond[, tzinfo]]]]])
```

All arguments are optional. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments may be ints or longs, in the following ranges:

- $0 \leq \text{hour} < 24$
- $0 \leq \text{minute} < 60$
- $0 \leq \text{second} < 60$
- $0 \leq \text{microsecond} < 1000000$.

If an argument outside those ranges is given, ValueError is raised. All default to 0 except tzinfo, which defaults to None.

Class attributes:

- **time.min**
  
  The earliest representable time, `time(0, 0, 0, 0)`.

- **time.max**
  
  The latest representable time, `time(23, 59, 59, 999999)`.

- **time.resolution**
  
  The smallest possible difference between non-equal time objects, `timedelta(microseconds=1)`, although note that arithmetic on time objects is not supported.

Instance attributes (read-only):
time.hour
In range(24).

time.minute
In range(60).

time.second
In range(60).

time.microsecond
In range(1000000).

time.tzinfo
The object passed as the tzinfo argument to the time constructor, or None if none was passed.

Supported operations:

- comparison of time to time, where a is considered less than b when a precedes b in time. If one comparand is naive and the other is aware, TypeError is raised. If both comparands are aware, and have the same tzinfo attribute, the common tzinfo attribute is ignored and the base times are compared. If both comparands are aware and have different tzinfo attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from self.utcoffset()). In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a time object is compared to an object of a different type, TypeError is raised unless the comparison is == or !=. The latter cases return False or True, respectively.
- hash, use as dict key
- efficient pickling
- in Boolean contexts, a time object is considered to be true if and only if, after converting it to minutes and subtracting utcoffset()
(or 0 if that's None), the result is non-zero.

Instance methods:

time.replace([hour[, minute[, second[, microsecond[, tzinfo]]]]])

Return a time with the same value, except for those attributes given new values by whichever keyword arguments are specified. Note that tzinfo=None can be specified to create a naive time from an aware time, without conversion of the time data.

time.isoformat()

Return a string representing the time in ISO 8601 format, HH:MM:SS.mmmmmm or, if self.microsecond is 0, HH:MM:SS If utcoffset() does not return None, a 6-character string is appended, giving the UTC offset in (signed) hours and minutes: HH:MM:SS.mmmmmm+HH:MM or, if self.microsecond is 0, HH:MM:SS+HH:MM

time.__str__()

For a time t, str(t) is equivalent to t.isoformat().

time.strftime(format)

Return a string representing the time, controlled by an explicit format string. For a complete list of formatting directives, see section strftime() and strptime() Behavior.

time.__format__(format)

Same as time.strftime(). This makes it possible to specify a format string for a time object when using str.format(). See section strftime() and strptime() Behavior.

time.utcoffset()

If tzinfo is None, returns None, else returns self.tzinfo.utcoffset(None), and raises an exception if the
latter doesn’t return None or a timedelta object representing a whole number of minutes with magnitude less than one day.

time.**dst**()

If **tzinfo** is None, returns None, else returns `self.tzinfo.dst(None)`, and raises an exception if the latter doesn’t return None, or a timedelta object representing a whole number of minutes with magnitude less than one day.

time.**tzname**()

If **tzinfo** is None, returns None, else returns `self.tzinfo.tzname(None)`, or raises an exception if the latter doesn’t return None or a string object.

Example:

```python
>>> from datetime import time, tzinfo
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self, dt):
...         return 'Europe/Prague'
...
>>> t = time(12, 10, 30, tzinfo=GMT1())
>>> t
datetime.time(12, 10, 30, tzinfo=<GMT1 object at 0x...>)
>>> gmt = GMT1()
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'Europe/Prague'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 Europe/Prague'
>>> 'The {} is {:H:%M}.format("time", t)
'The time is 12:10.'
```
8.1.6. `tzinfo` Objects

class `datetime.tzinfo`

This is an abstract base class, meaning that this class should not be instantiated directly. You need to derive a concrete subclass, and (at least) supply implementations of the standard `tzinfo` methods needed by the `datetime` methods you use. The `datetime` module does not supply any concrete subclasses of `tzinfo`.

An instance of (a concrete subclass of) `tzinfo` can be passed to the constructors for `datetime` and `time` objects. The latter objects view their attributes as being in local time, and the `tzinfo` object supports methods revealing offset of local time from UTC, the name of the time zone, and DST offset, all relative to a date or time object passed to them.

Special requirement for pickling: A `tzinfo` subclass must have an `__init__()` method that can be called with no arguments, else it can be pickled but possibly not unpickled again. This is a technical requirement that may be relaxed in the future.

A concrete subclass of `tzinfo` may need to implement the following methods. Exactly which methods are needed depends on the uses made of aware `datetime` objects. If in doubt, simply implement all of them.

`tzinfo.utcoffset(self, dt)`

Return offset of local time from UTC, in minutes east of UTC. If local time is west of UTC, this should be negative. Note that this is intended to be the total offset from UTC; for example, if a `tzinfo` object represents both time zone and DST adjustments,
The `utcoffset()` function should return their sum. If the UTC offset isn't known, return `None`. Else the value returned must be a `timedelta` object specifying a whole number of minutes in the range -1439 to 1439 inclusive (1440 = 24*60; the magnitude of the offset must be less than one day). Most implementations of `utcoffset()` will probably look like one of these two:

```
return CONSTANT # fixed-offset class
return CONSTANT + self.dst(dt) # daylight-aware class
```

If `utcoffset()` does not return `None`, `dst()` should not return `None` either.

The default implementation of `utcoffset()` raises `NotImplementedError`.

`tzinfo.dst(self, dt)`

Return the daylight saving time (DST) adjustment, in minutes east of UTC, or `None` if DST information isn't known. Return `timedelta(0)` if DST is not in effect. If DST is in effect, return the offset as a `timedelta` object (see `utcoffset()` for details). Note that DST offset, if applicable, has already been added to the UTC offset returned by `utcoffset()`, so there's no need to consult `dst()` unless you're interested in obtaining DST info separately. For example, `datetime.timetuple()` calls its `tzinfo` attribute's `dst()` method to determine how the `tm_isdst` flag should be set, and `tzinfo.fromutc()` calls `dst()` to account for DST changes when crossing time zones.

An instance `tz` of a `tzinfo` subclass that models both standard and daylight times must be consistent in this sense:

```
tz.utcoffset(dt) - tz.dst(dt)
```
must return the same result for every `datetime dt` with `dt.tzinfo == tz` For sane `tzinfo` subclasses, this expression yields the time zone’s “standard offset”, which should not depend on the date or the time, but only on geographic location. The implementation of `datetime.astimezone()` relies on this, but cannot detect violations; it’s the programmer’s responsibility to ensure it. If a `tzinfo` subclass cannot guarantee this, it may be able to override the default implementation of `tzinfo.fromutc()` to work correctly with `astimezone()` regardless.

Most implementations of `dst()` will probably look like one of these two:

```python
def dst(self, dt):
    # a fixed-offset class: doesn't account for DST
    return timedelta(0)
```
or

```python
def dst(self, dt):
    # Code to set dston and dstoff to the time zone's DST
    # transition times based on the input dt.year, and express
    # in standard local time. Then

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

The default implementation of `dst()` raises `NotImplementedError`.

`tzinfo.tzname(self, dt)`

Return the time zone name corresponding to the `datetime` object `dt`, as a string. Nothing about string names is defined by the `datetime` module, and there’s no requirement that it mean anything in particular. For example, “GMT”, “UTC”, “-500”, “-5:00”,

```
“EDT”, “US/Eastern”, “America/New York” are all valid replies. Return None if a string name isn't known. Note that this is a method rather than a fixed string primarily because some tzinfo subclasses will wish to return different names depending on the specific value of dt passed, especially if the tzinfo class is accounting for daylight time.

The default implementation of tzname() raises NotImplementedError.

These methods are called by a datetime or time object, in response to their methods of the same names. A datetime object passes itself as the argument, and a time object passes None as the argument. A tzinfo subclass’s methods should therefore be prepared to accept a dt argument of None, or of class datetime.

When None is passed, it’s up to the class designer to decide the best response. For example, returning None is appropriate if the class wishes to say that time objects don’t participate in the tzinfo protocols. It may be more useful for utcoffset(None) to return the standard UTC offset, as there is no other convention for discovering the standard offset.

When a datetime object is passed in response to a datetime method, dt.tzinfo is the same object as self. tzinfo methods can rely on this, unless user code calls tzinfo methods directly. The intent is that the tzinfo methods interpret dt as being in local time, and not need worry about objects in other timezones.

There is one more tzinfo method that a subclass may wish to override:

tzinfo.fromutc(self, dt)
This is called from the default `datetime.astimezone()` implementation. When called from that, `dt.tzinfo` is `self`, and `dt`'s date and time data are to be viewed as expressing a UTC time. The purpose of `fromutc()` is to adjust the date and time data, returning an equivalent datetime in `self`'s local time.

Most `tzinfo` subclasses should be able to inherit the default `fromutc()` implementation without problems. It's strong enough to handle fixed-offset time zones, and time zones accounting for both standard and daylight time, and the latter even if the DST transition times differ in different years. An example of a time zone the default `fromutc()` implementation may not handle correctly in all cases is one where the standard offset (from UTC) depends on the specific date and time passed, which can happen for political reasons. The default implementations of `astimezone()` and `fromutc()` may not produce the result you want if the result is one of the hours straddling the moment the standard offset changes.

Skipping code for error cases, the default `fromutc()` implementation acts like:

```python
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst  # this is self's standard offset
    if delta:
        dt += delta  # convert to standard local time
        dtdst = dt.dst()  # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```
Example `tzinfo` classes:

```python
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)

# A UTC class.

class UTC(tzinfo):
    """UTC""

    def utcoffset(self, dt):
        return ZERO

    def tzname(self, dt):
        return "UTC"

    def dst(self, dt):
        return ZERO

utc = UTC()

# A class building tzinfo objects for fixed-offset time zones.
# Note that FixedOffset(0, "UTC") is a different way to build a
# UTC tzinfo object.

class FixedOffset(tzinfo):
    """Fixed offset in minutes east from UTC.""

    def __init__(self, offset, name):
        self.__offset = timedelta(minutes = offset)
        self.__name = name

    def utcoffset(self, dt):
        return self.__offset

    def tzname(self, dt):
        return self.__name

    def dst(self, dt):
        return ZERO

# A class capturing the platform's idea of local time.

import time as _time
```
STDOFFSET = timedelta(seconds = __time.timezone)
if __time.daylight:
    DSTOFFSET = timedelta(seconds = __time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):
    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

def dst(self, dt):
    if self._isdst(dt):
        return DSTDIFF
    else:
        return ZERO

def tzname(self, dt):
    return __time.tzname[self._isdst(dt)]

def _isdst(self, dt):
    tt = (__time.year, __time.month, __time.day,
        __time.hour, __time.minute, __time.second,
        __time.weekday(), 0, 0)
    stamp = __time.mktime(tt)
    tt = __time.localtime(stamp)
    return tt.tm_isdst > 0

Local = LocalTimezone()

# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt

# US DST Rules
# This is a simplified (i.e., wrong for a few cases) set of rules for US DST start and end times. For a complete and up-to-date set of rules and timezone definitions, visit the Olson Database (or try pytz):
# http://www.twinsun.com/tz/tz-link.htm
# http://sourceforge.net/projects/pytz/ (might not be up-to-date)

# In the US, since 2007, DST starts at 2am (standard time) on the first Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time; 1am standard time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 1)
# From 1987 to 2006, DST used to start at 2am (standard time) on Sunday in April and to end at 2am (DST time; 1am standard time) on the first Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 1, 1)
# From 1967 to 1986, DST used to start at 2am (standard time) on Sunday in April (the one on or after April 24) and to end at 2am (DST time; 1am standard time) on the last Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)

class USTimeZone(tzinfo):
    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname

    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)

    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both
# It depends on how you want to treat them. The default fromutc() implementation (called by the default_astimezone() implementation) passes a datetime with dt.tzinfo.

```python
return ZERO
assert dt.tzinfo is self
```

# Find start and end times for US DST. For years before 2006, return ZERO for no DST.

```python
if 2006 < dt.year:
    dststart, dstend = DSTSTART_2007, DSTEND_2007
elif 1986 < dt.year < 2007:
elif 1966 < dt.year < 1987:
else:
    return ZERO
```

start = first_sunday_on_or_after(dststart.replace(year=dt.year))
end = first_sunday_on_or_after(dstend.replace(year=dt.year))

# Can't compare naive to aware objects, so strip the timezone from dt first.

```python
if start <= dt.replace(tzinfo=None) < end:
    return HOUR
else:
    return ZERO
```

Eastern = USTimeZone(-5, "Eastern", "EST", "EDT")
Central = USTimeZone(-6, "Central", "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific = USTimeZone(-8, "Pacific", "PST", "PDT")

Note that there are unavoidable subtleties twice per year in a tzinfo subclass accounting for both standard and daylight time, at the DST transition points. For concreteness, consider US Eastern (UTC -0500), where EDT begins the minute after 1:59 (EST) on the second Sunday in March, and ends the minute after 1:59 (EDT) on the first Sunday in November:

|-----|------|------|------|------|------|------|
When DST starts (the “start” line), the local wall clock leaps from 1:59 to 3:00. A wall time of the form 2:MM doesn’t really make sense on that day, so \texttt{astimezone(Eastern)} won’t deliver a result with \texttt{hour} \texttt{== 2} on the day DST begins. In order for \texttt{astimezone()} to make this guarantee, the \texttt{tzinfo.dst()} method must consider times in the “missing hour” (2:MM for Eastern) to be in daylight time.

When DST ends (the “end” line), there’s a potentially worse problem: there’s an hour that can’t be spelled unambiguously in local wall time: the last hour of daylight time. In Eastern, that’s times of the form 5:MM UTC on the day daylight time ends. The local wall clock leaps from 1:59 (daylight time) back to 1:00 (standard time) again. Local times of the form 1:MM are ambiguous. \texttt{astimezone()} mimics the local clock’s behavior by mapping two adjacent UTC hours into the same local hour then. In the Eastern example, UTC times of the form 5:MM and 6:MM both map to 1:MM when converted to Eastern. In order for \texttt{astimezone()} to make this guarantee, the \texttt{tzinfo.dst()} method must consider times in the “repeated hour” to be in standard time. This is easily arranged, as in the example, by expressing DST switch times in the time zone’s standard local time.

Applications that can’t bear such ambiguities should avoid using hybrid \texttt{tzinfo} subclasses; there are no ambiguities when using UTC, or any other fixed-offset \texttt{tzinfo} subclass (such as a class representing only EST (fixed offset -5 hours), or only EDT (fixed offset -4 hours)).

\textbf{See also:}

\texttt{pytz}

The standard library has no \texttt{tzinfo} instances, but there exists a
third-party library which brings the IANA timezone database (also known as the Olson database) to Python: pytz.

pytz contains up-to-date information and its usage is recommended.

IANA timezone database
The Time Zone Database (often called tz or zoneinfo) contains code and data that represent the history of local time for many representative locations around the globe. It is updated periodically to reflect changes made by political bodies to time zone boundaries, UTC offsets, and daylight-saving rules.
8.1.7. strftime() and strptime() Behavior

date, datetime, and time objects all support a strftime(format) method, to create a string representing the time under the control of an explicit format string. Broadly speaking, d.strftime(fmt) acts like the time module’s time.strftime(fmt, d.timetuple()) although not all objects support a timetuple() method.

Conversely, the datetime.strptime() class method creates a datetime object from a string representing a date and time and a corresponding format string. datetime.strptime(date_string, format) is equivalent to datetime(*(time.strptime(date_string, format)[0:6])).

For time objects, the format codes for year, month, and day should not be used, as time objects have no such values. If they’re used anyway, 1900 is substituted for the year, and 1 for the month and day.

For date objects, the format codes for hours, minutes, seconds, and microseconds should not be used, as date objects have no such values. If they’re used anyway, 0 is substituted for them.

The full set of format codes supported varies across platforms, because Python calls the platform C library’s strftime() function, and platform variations are common. To see the full set of format codes supported on your platform, consult the strftime(3) documentation.

The following is a list of all the format codes that the C standard (1989 version) requires, and these work on all platforms with a standard C implementation. Note that the 1999 version of the C
standard added additional format codes.

The exact range of years for which `strftime()` works also varies across platforms. Regardless of platform, years before 1900 cannot be used.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Meaning</th>
<th>Example</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Weekday as locale’s abbreviated name.</td>
<td>Sun, Mon, ..., Sat (en_US); So, Mo, ..., Sa (de_DE)</td>
<td>(1)</td>
</tr>
<tr>
<td>%A</td>
<td>Weekday as locale’s full name.</td>
<td>Sunday, Monday, ..., Saturday (en_US); Sonntag, Montag, ..., Samstag (de_DE)</td>
<td>(1)</td>
</tr>
<tr>
<td>%w</td>
<td>Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.</td>
<td>0, 1, ..., 6</td>
<td></td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month as a zero-padded decimal number.</td>
<td>01, 02, ..., 31</td>
<td></td>
</tr>
<tr>
<td>%b</td>
<td>Month as locale’s abbreviated name.</td>
<td>Jan, Feb, ..., Dec (en_US); Jan, Feb, ..., Dez (de_DE)</td>
<td>(1)</td>
</tr>
<tr>
<td>%B</td>
<td>Month as locale’s full name.</td>
<td>January, February, ..., December (en_US); Januar, Februar, ..., Dezember (de_DE)</td>
<td>(1)</td>
</tr>
<tr>
<td>%m</td>
<td>Month as a zero-padded decimal number.</td>
<td>01, 02, ..., 12</td>
<td></td>
</tr>
<tr>
<td>%y</td>
<td>Year without century as a zero-padded decimal</td>
<td>00, 01, ..., 99</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Description</td>
<td>Examples</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td><code>%H</code></td>
<td>Hour (24-hour clock) as a zero-padded decimal number.</td>
<td>00, 01, ..., 23</td>
<td></td>
</tr>
<tr>
<td><code>%I</code></td>
<td>Hour (12-hour clock) as a zero-padded decimal number.</td>
<td>01, 02, ..., 12</td>
<td></td>
</tr>
<tr>
<td><code>%p</code></td>
<td>Locale's equivalent of either AM or PM.</td>
<td>AM, PM (en_US); am, pm (de_DE)</td>
<td>1), (2)</td>
</tr>
<tr>
<td><code>%M</code></td>
<td>Minute as a zero-padded decimal number.</td>
<td>00, 01, ..., 59</td>
<td></td>
</tr>
<tr>
<td><code>%S</code></td>
<td>Second as a zero-padded decimal number.</td>
<td>00, 01, ..., 59</td>
<td>(3)</td>
</tr>
<tr>
<td><code>%f</code></td>
<td>Microsecond as a decimal number, zero-padded on the left.</td>
<td>0000000, 000001, ..., 999999</td>
<td>(4)</td>
</tr>
<tr>
<td><code>%z</code></td>
<td>UTC offset in the form +HHMM or -HHMM (empty string if the object is naive).</td>
<td>(empty), +0000, -0400, +1030</td>
<td>(5)</td>
</tr>
<tr>
<td><code>%Z</code></td>
<td>Time zone name (empty string if the object is naive).</td>
<td>(empty), UTC, EST, CST</td>
<td></td>
</tr>
<tr>
<td><code>%j</code></td>
<td>Day of the year as a zero-padded decimal number.</td>
<td>001, 002, ..., 366</td>
<td></td>
</tr>
<tr>
<td><code>%U</code></td>
<td>Week number of the year (Sunday as the first day of the week) as a zero padded decimal number. All days in a new year preceding the first Sunday are considered to be in week 0.</td>
<td>00, 01, ..., 53</td>
<td>(6)</td>
</tr>
<tr>
<td></td>
<td>Week number of the year (Monday as the first day of</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
%W the week) as a decimal number. All days in a new year preceding the first Monday are considered to be in week 0. 00, 01, ..., 53 (6)

%c Locale's appropriate date and time representation. Tue Aug 16 21:30:00 1988 (en_US); Di 16 Aug 21:30:00 1988 (de_DE) (1)

%x Locale's appropriate date representation. 08/16/88 (None); 08/16/1988 (en_US); 16.08.1988 (de_DE) (1)

%X Locale's appropriate time representation. 21:30:00 (en_US); 21:30:00 (de_DE) (1)

%% A literal '%' character. %

Notes:

1. Because the format depends on the current locale, care should be taken when making assumptions about the output value. Field orderings will vary (for example, “month/day/year” versus “day/month/year”), and the output may contain Unicode characters encoded using the locale’s default encoding (for example, if the current locale is ja_JP, the default encoding could be any one of eucJP, SJIS, or utf-8; use locale.getlocale() to determine the current locale’s encoding).

2. When used with the strftime() method, the %p directive only affects the output hour field if the %I directive is used to parse the hour.

3. Unlike the time module, the datetime module does not support leap seconds.
4. `%f` is an extension to the set of format characters in the C standard (but implemented separately in datetime objects, and therefore always available). When used with the `strptime()` method, the `%f` directive accepts from one to six digits and zero pads on the right.

*New in version 2.6.*

5. For a naive object, the `%z` and `%Z` format codes are replaced by empty strings.

For an aware object:

```%z```

`utcoffset()` is transformed into a 5-character string of the form `+HHMM` or `-HHMM`, where HH is a 2-digit string giving the number of UTC offset hours, and MM is a 2-digit string giving the number of UTC offset minutes. For example, if `utcoffset()` returns `timedelta(hours=-3, minutes=-30)`, `%z` is replaced with the string `'-0330'`.

```%Z```

If `tzname()` returns `None`, `%Z` is replaced by an empty string. Otherwise `%Z` is replaced by the returned value, which must be a string.

6. When used with the `strptime()` method, `%U` and `%W` are only used in calculations when the day of the week and the year are specified.

Footnotes

[1] If, that is, we ignore the effects of Relativity
Standard Library » 8. Data Types »
8.2. calendar — General calendar-related functions

Source code: Lib/calendar.py

This module allows you to output calendars like the Unix `cal` program, and provides additional useful functions related to the calendar. By default, these calendars have Monday as the first day of the week, and Sunday as the last (the European convention). Use `setfirstweekday()` to set the first day of the week to Sunday (6) or to any other weekday. Parameters that specify dates are given as integers. For related functionality, see also the `datetime` and `time` modules.

Most of these functions and classes rely on the `datetime` module which uses an idealized calendar, the current Gregorian calendar indefinitely extended in both directions. This matches the definition of the “proleptic Gregorian” calendar in Dershowitz and Reingold’s book “Calendrical Calculations”, where it’s the base calendar for all computations.

```python
class calendar.Calendar([firstweekday])
```

Creates a `Calendar` object. `firstweekday` is an integer specifying the first day of the week. 0 is Monday (the default), 6 is Sunday.

A `Calendar` object provides several methods that can be used for preparing the calendar data for formatting. This class doesn’t do any formatting itself. This is the job of subclasses.

*New in version 2.5.*

`Calendar` instances have the following methods:
iterweekdays()
Return an iterator for the week day numbers that will be used for one week. The first value from the iterator will be the same as the value of the firstweekday property.

itermonthdates(year, month)
Return an iterator for the month month (1–12) in the year year. This iterator will return all days (as datetime.date objects) for the month and all days before the start of the month or after the end of the month that are required to get a complete week.

itermonthdays2(year, month)
Return an iterator for the month month in the year year similar to itermonthdates(). Days returned will be tuples consisting of a day number and a week day number.

itermonthdays(year, month)
Return an iterator for the month month in the year year similar to itermonthdates(). Days returned will simply be day numbers.

monthdatescalendar(year, month)
Return a list of the weeks in the month month of the year as full weeks. Weeks are lists of seven datetime.date objects.

monthdays2calendar(year, month)
Return a list of the weeks in the month month of the year as full weeks. Weeks are lists of seven tuples of day numbers and weekday numbers.

monthdayscalendar(year, month)
Return a list of the weeks in the month month of the year as full weeks. Weeks are lists of seven day numbers.
yeardatescalendar\,(year[, width])
Return the data for the specified year ready for formatting. The return value is a list of month rows. Each month row contains up to width months (defaulting to 3). Each month contains between 4 and 6 weeks and each week contains 1–7 days. Days are \texttt{datetime.date} objects.

yeardays2calendar\,(year[, width])
Return the data for the specified year ready for formatting (similar to \texttt{yeardatescalendar()}). Entries in the week lists are tuples of day numbers and weekday numbers. Day numbers outside this month are zero.

eyeardayscalendar\,(year[, width])
Return the data for the specified year ready for formatting (similar to \texttt{yeardatescalendar()}). Entries in the week lists are day numbers. Day numbers outside this month are zero.

class calendar.\texttt{TextCalendar}(\{firstweekday\})
This class can be used to generate plain text calendars.

\textit{New in version 2.5.}

\texttt{TextCalendar} instances have the following methods:

\texttt{formatmonth(theyear, themonth[, w[, l]])}
Return a month's calendar in a multi-line string. If w is provided, it specifies the width of the date columns, which are centered. If l is given, it specifies the number of lines that each week will use. Depends on the first weekday as specified in the constructor or set by the \texttt{setfirstweekday()} method.
\textbf{prmonth}(theyear, themonth[, w[, l]])

Print a month’s calendar as returned by \texttt{formatmonth()}.

\textbf{formatyear}(theyear[, w[, l[, c[, m]]]])

Return a \textit{m}-column calendar for an entire year as a multi-line string. Optional parameters \textit{w}, \textit{l}, and \textit{c} are for date column width, lines per week, and number of spaces between month columns, respectively. Depends on the first weekday as specified in the constructor or set by the \texttt{setfirstweekday()} method. The earliest year for which a calendar can be generated is platform-dependent.

\textbf{pryear}(theyear[, w[, l[, c[, m]]]])

Print the calendar for an entire year as returned by \texttt{formatyear()}.

\textit{class} \texttt{calendar.HTMLCalendar}([\textit{firstweekday}])

This class can be used to generate HTML calendars.

\textit{New in version 2.5.}

\texttt{HTMLCalendar} instances have the following methods:

\textbf{formatmonth}(theyear, themonth[, withyear])

Return a month’s calendar as an HTML table. If \textit{withyear} is true the year will be included in the header, otherwise just the month name will be used.

\textbf{formatyear}(theyear[, width])

Return a year’s calendar as an HTML table. \textit{width} (defaulting to 3) specifies the number of months per row.

\textbf{formatyearpage}(theyear[, width[, css[, encoding]]])
Return a year’s calendar as a complete HTML page. `width` (defaulting to 3) specifies the number of months per row. `css` is the name for the cascading style sheet to be used. `None` can be passed if no style sheet should be used. `encoding` specifies the encoding to be used for the output (defaulting to the system default encoding).

```python
class calendar.LocaleTextCalendar([firstweekday[, locale]])
```

This subclass of `TextCalendar` can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.

*New in version 2.5.*

```python
class calendar.LocaleHTMLCalendar([firstweekday[, locale]])
```

This subclass of `HTMLCalendar` can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.

*New in version 2.5.*

**Note:** The `formatweekday()` and `formatmonthname()` methods of these two classes temporarily change the current locale to the given `locale`. Because the current locale is a process-wide setting, they are not thread-safe.

For simple text calendars this module provides the following functions.

```python
calendar.setfirstweekday(weekday)
```
Sets the weekday (0 is Monday, 6 is Sunday) to start each week. The values `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, and `SUNDAY` are provided for convenience. For example, to set the first weekday to Sunday:

```python
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

*New in version 2.0.*

**calendar.** `firstweekday()`

Returns the current setting for the weekday to start each week.

*New in version 2.0.*

**calendar.** `isleap(year)`

Returns `True` if `year` is a leap year, otherwise `False`.

**calendar.** `leapdays(y1, y2)`

Returns the number of leap years in the range from `y1` to `y2` (exclusive), where `y1` and `y2` are years.

*Changed in version 2.0:* This function didn’t work for ranges spanning a century change in Python 1.5.2.

**calendar.** `weekday(year, month, day)`

Returns the day of the week (0 is Monday) for `year` (1970—...), `month` (1–12), `day` (1–31).

**calendar.** `weekheader(n)`

Return a header containing abbreviated weekday names. `n` specifies the width in characters for one weekday.

**calendar.** `monthrange(year, month)`

Returns weekday of first day of the month and number of days in month, for the specified `year` and `month`. 
calendar.monthcalendar(year, month)
    Returns a matrix representing a month's calendar. Each row represents a week; days outside of the month a represented by zeros. Each week begins with Monday unless set by setfirstweekday().

calendar.prmont(theyear, themonth[, w[, l]])
    Prints a month's calendar as returned by month().

calendar.month(theyear, themonth[, w[, l]])
    Returns a month's calendar in a multi-line string using the formatmonth() of the TextCalendar class.

    New in version 2.0.

calendar.prcal(year[, w[, l[c]]])
    Prints the calendar for an entire year as returned by calendar().

calendar.calendar(year[, w[, l[c]]])
    Returns a 3-column calendar for an entire year as a multi-line string using the formatyear() of the TextCalendar class.

    New in version 2.0.

calendar.timegm(tuple)
    An unrelated but handy function that takes a time tuple such as returned by the gmtime() function in the time module, and returns the corresponding Unix timestamp value, assuming an epoch of 1970, and the POSIX encoding. In fact, time.gmtime() and timegm() are each others' inverse.

    New in version 2.0.

The calendar module exports the following data attributes:
calendar\texttt{.day\_name}

An array that represents the days of the week in the current locale.

calendar\texttt{.day\_abbr}

An array that represents the abbreviated days of the week in the current locale.

calendar\texttt{.month\_name}

An array that represents the months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and \texttt{month\_name[0]} is the empty string.

calendar\texttt{.month\_abbr}

An array that represents the abbreviated months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and \texttt{month\_abbr[0]} is the empty string.

See also:

\textbf{Module} \texttt{datetime}

Object-oriented interface to dates and times with similar functionality to the \texttt{time} module.

\textbf{Module} \texttt{time}

Low-level time related functions.
8.3. collections — High-performance container datatypes

New in version 2.4.

Source code: Lib/collections.py and Lib/_abcoll.py

This module implements specialized container datatypes providing alternatives to Python’s general purpose built-in containers, dict, list, set, and tuple.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>namedtuple()</td>
<td>factory function for creating tuple subclasses with named fields</td>
<td>New in version 2.6.</td>
</tr>
<tr>
<td>deque</td>
<td>list-like container with fast appends and pops on either end</td>
<td>New in version 2.4.</td>
</tr>
<tr>
<td>Counter</td>
<td>dict subclass for counting hashable objects</td>
<td>New in version 2.7.</td>
</tr>
<tr>
<td>OrderedDict</td>
<td>dict subclass that remembers the order entries were added</td>
<td>New in version 2.7.</td>
</tr>
<tr>
<td>defaultdict</td>
<td>dict subclass that calls a factory function to supply missing values</td>
<td>New in version 2.5.</td>
</tr>
</tbody>
</table>

In addition to the concrete container classes, the collections module
provides abstract base classes that can be used to test whether a class provides a particular interface, for example, whether it is hashable or a mapping.
8.3.1. counter objects

A counter tool is provided to support convenient and rapid tallies. For example:

```python
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})
```

```python
>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 6), ('you', 554), ('a', 546), ('my', 514), ('hamlet', 471), ('in'
```

```

class collections.Counter([iterable-or-mapping])

A Counter is a dict subclass for counting hashable objects. It is an unordered collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The Counter class is similar to bags or multisets in other languages.

Elements are counted from an iterable or initialized from another mapping (or counter):

```python
>>> c = Counter()  # a new, empty c
>>> c = Counter('gallahad')  # a new counter
>>> c = Counter({'red': 4, 'blue': 2})  # a new counter
>>> c = Counter(cats=4, dogs=8)  # a new counter
```

Counter objects have a dictionary interface except that they
return a zero count for missing items instead of raising a

\textbf{KeyError}:

\begin{verbatim}
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']
0  # count of a missing element

Setting a count to zero does not remove an element from a
counter. Use \texttt{del} to remove it entirely:

\begin{verbatim}
>>> c['sausage'] = 0  # counter entry with zero count
>>> del c['sausage']  # del actually removes the entry

\textit{New in version 2.7.}

Counter objects support three methods beyond those available
for all dictionaries:

\texttt{elements()}

Return an iterator over elements repeating each as many
times as its count. Elements are returned in arbitrary order. If
an element’s count is less than one, \texttt{elements()} will ignore it.

\begin{verbatim}
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']

\texttt{most\_common([n])}

Return a list of the \textit{n} most common elements and their counts
from the most common to the least. If \textit{n} is omitted or \texttt{None},
\texttt{most\_common()} returns \textit{all} elements in the counter. Elements
with equal counts are ordered arbitrarily:

\begin{verbatim}
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('r', 2), ('b', 2)]
\end{verbatim}
\end{verbatim}
subtract([iterable-or-mapping])

Elements are subtracted from an iterable or from another mapping (or counter). Like `dict.update()` but subtracts counts instead of replacing them. Both inputs and outputs may be zero or negative.

```python
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

The usual dictionary methods are available for `Counter` objects except for two which work differently for counters.

fromkeys(iterable)

This class method is not implemented for `Counter` objects.

update([iterable-or-mapping])

Elements are counted from an iterable or added-in from another mapping (or counter). Like `dict.update()` but adds counts instead of replacing them. Also, the iterable is expected to be a sequence of elements, not a sequence of (key, value) pairs.

Common patterns for working with `Counter` objects:

```python
sum(c.values()) # total of all counts
c.clear() # reset all counts
list(c) # list unique elements
set(c) # convert to a set
dict(c) # convert to a regular dictionary
list(c.items()) # convert to a list of (elem, c)
Counter(dict(list_of_pairs)) # convert from a list of (elem, c)
c.most_common()[:n:-1] # n least common elements
c += Counter() # remove zero and negative counts
```
Several mathematical operations are provided for combining `Counter` objects to produce multisets (counters that have counts greater than zero). Addition and subtraction combine counters by adding or subtracting the counts of corresponding elements. Intersection and union return the minimum and maximum of corresponding counts. Each operation can accept inputs with signed counts, but the output will exclude results with counts of zero or less.

```python
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d  # add two counters together: c + d
Counter({'a': 4, 'b': 3})
>>> c - d  # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d  # intersection: min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d  # union: max(c[x], d[x])
Counter({'a': 3, 'b': 2})
```

**Note:** Counters were primarily designed to work with positive integers to represent running counts; however, care was taken to not unnecessarily preclude use cases needing other types or negative values. To help with those use cases, this section documents the minimum range and type restrictions.

- The `Counter` class itself is a dictionary subclass with no restrictions on its keys and values. The values are intended to be numbers representing counts, but you could store anything in the value field.
- The `most_common()` method requires only that the values be orderable.
- For in-place operations such as `c[key] += 1`, the value type need only support addition and subtraction. So fractions, floats, and decimals would work and negative values are supported. The same is also true for `update()` and `subtract()` which allow negative and zero values for both inputs and
The multiset methods are designed only for use cases with positive values. The inputs may be negative or zero, but only outputs with positive values are created. There are no type restrictions, but the value type needs to support addition, subtraction, and comparison.

- The `elements()` method requires integer counts. It ignores zero and negative counts.

See also:

- Counter class adapted for Python 2.5 and an early Bag recipe for Python 2.4.
- in Smalltalk.
- Wikipedia entry for Multisets.
- C++ multisets tutorial with examples.
- For mathematical operations on multisets and their use cases, see Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19.
- To enumerate all distinct multisets of a given size over a given set of elements, see `itertools.combinations_with_replacement()`.

```
map(Counter, combinations_with_replacement('ABC', 2))
→ AA AB AC BB BC CC
```
class collections.deque([iterable[, maxlen]])

Returns a new deque object initialized left-to-right (using append()) with data from iterable. If iterable is not specified, the new deque is empty.

Deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction.

Though list objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movement costs for pop(0) and insert(0, v) operations which change both the size and position of the underlying data representation.

New in version 2.4.

If maxlen is not specified or is None, deques may grow to an arbitrary length. Otherwise, the deque is bounded to the specified maximum length. Once a bounded length deque is full, when new items are added, a corresponding number of items are discarded from the opposite end. Bounded length deques provide functionality similar to the tail filter in Unix. They are also useful for tracking transactions and other pools of data where only the most recent activity is of interest.

Changed in version 2.6: Added maxlen parameter.

Deque objects support the following methods:
**append**(\(x\))
Add \(x\) to the right side of the deque.

**appendleft**(\(x\))
Add \(x\) to the left side of the deque.

**clear**()
Remove all elements from the deque leaving it with length 0.

**count**(\(x\))
Count the number of deque elements equal to \(x\).

*New in version 2.7.*

**extend**(iterable)
Extend the right side of the deque by appending elements from the iterable argument.

**extendleft**(iterable)
Extend the left side of the deque by appending elements from iterable. Note, the series of left appends results in reversing the order of elements in the iterable argument.

**pop**()
Remove and return an element from the right side of the deque. If no elements are present, raises an **IndexError**.

**popleft**()
Remove and return an element from the left side of the deque. If no elements are present, raises an **IndexError**.

**remove**(value)
Removed the first occurrence of value. If not found, raises a **ValueError**.
New in version 2.5.

reverse()
Reverse the elements of the deque in-place and then return None.

New in version 2.7.

rotate(n)
Rotate the deque n steps to the right. If n is negative, rotate to the left. Rotating one step to the right is equivalent to:
\[ \text{d}\.\text{appendleft}(\text{d}\text{.pop()}). \]

Deque objects also provide one read-only attribute:

maxlen
Maximum size of a deque or None if unbounded.

New in version 2.7.

In addition to the above, deques support iteration, pickling, len(d), reversed(d), copy.copy(d), copy.deepcopy(d), membership testing with the in operator, and subscript references such as \[ d[-1] \]. Indexed access is O(1) at both ends but slows to O(n) in the middle. For fast random access, use lists instead.

Example:

```python
>>> from collections import deque
>>> d = deque('ghi')    # make a new deque with the
>>> for elem in d:       # iterate over the deque's
...    print elem.upper()  # make new entry to the right
G
H
I
>>> d.append('j')       # add new entry to the right
>>> d.appendleft('f')   # add new entry to the left
```
>>> d = deque(['f', 'g', 'h', 'i', 'j'])  # show the representation
>>> d.pop()  # return and remove the rightmost item
'j'
>>> d.popleft()  # return and remove the leftmost item
'f'
>>> list(d)  # list the contents of the deque
['g', 'h', 'i']
>>> d[0]  # peek at leftmost item
'g'
>>> d[-1]  # peek at rightmost item
'i'

>>> list(reversed(d))  # list the contents of a deque in reverse order
['i', 'h', 'g']
>>> 'h' in d  # search the deque
True

>>> d.extend('jk1')  # add multiple elements at once
>>> d  # show the updated deque
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> d.rotate(1)  # right rotation
>>> d  # show the rotated deque
deque(['l', 'g', 'h', 'i', 'j', 'k'])

>>> d.rotate(-1)  # left rotation
>>> d  # show the left-rotated deque
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))  # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])

>>> d.clear()  # empty the deque
>>> d.pop()  # cannot pop from an empty deque
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')  # extendleft() reverses the order
>>> d  # show the extended deque
deque(['c', 'b', 'a'])

8.3.2.1. deque Recipes

This section shows various approaches to working with deques.
Bounded length deques provide functionality similar to the `tail` filter in Unix:

```python
def tail(filename, n=10):
    'Return the last n lines of a file'
    return deque(open(filename), n)
```

Another approach to using deques is to maintain a sequence of recently added elements by appending to the right and popping to the left:

```python
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45
    # http://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / float(n)
```

The `rotate()` method provides a way to implement `deque` slicing and deletion. For example, a pure Python implementation of `del d[n]` relies on the `rotate()` method to position elements to be popped:

```python
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

To implement `deque` slicing, use a similar approach applying `rotate()` to bring a target element to the left side of the deque. Remove old entries with `popleft()`, add new entries with `extend()`, and then reverse the rotation. With minor variations on that approach, it is easy to implement Forth style stack manipulations
such as dup, drop, swap, over, pick, rot, and roll.
8.3.3. defaultdict objects

```python
class collections.defaultdict([default_factory[, ...]])
```

Returns a new dictionary-like object. `defaultdict` is a subclass of the built-in `dict` class. It overrides one method and adds one writable instance variable. The remaining functionality is the same as for the `dict` class and is not documented here.

The first argument provides the initial value for the `default_factory` attribute; it defaults to `None`. All remaining arguments are treated the same as if they were passed to the `dict` constructor, including keyword arguments.

*New in version 2.5.*

`defaultdict` objects support the following method in addition to the standard `dict` operations:

```python
__missing__(key)
```

If the `default_factory` attribute is `None`, this raises a `KeyError` exception with the `key` as argument.

If `default_factory` is not `None`, it is called without arguments to provide a default value for the given `key`, this value is inserted in the dictionary for the `key`, and returned.

If calling `default_factory` raises an exception this exception is propagated unchanged.

This method is called by the `__getitem__()` method of the `dict` class when the requested key is not found; whatever it returns or raises is then returned or raised by `__getitem__()`.
Note that \texttt{\_\_missing\_()} is \textit{not} called for any operations besides \texttt{\_\_getitem\_()}. This means that \texttt{get()} will, like normal dictionaries, return \texttt{None} as a default rather than using \texttt{default\_factory}.

\texttt{defaultdict} objects support the following instance variable:

\textbf{default\_factory}

This attribute is used by the \texttt{\_\_missing\_()} method; it is initialized from the first argument to the constructor, if present, or to \texttt{None}, if absent.

\textbf{8.3.3.1. defaultdict Examples}

Using \texttt{list} as the \texttt{default\_factory}, it is easy to group a sequence of key-value pairs into a dictionary of lists:

```python
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4)
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...     ...
>>> d.items()
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

When each key is encountered for the first time, it is not already in the mapping; so an entry is automatically created using the \texttt{default\_factory} function which returns an empty \texttt{list}. The \texttt{list.append()} operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that key) and the \texttt{list.append()} operation adds another value to the list. This technique is simpler and faster than an equivalent technique using \texttt{dict.setdefault()}:

```python
>>> d = {}
```
for k, v in s:
    ... d.setdefault(k, []).append(v)
...

d.items()
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]

Setting the `default_factory` to `int` makes the `defaultdict` useful for counting (like a bag or multiset in other languages):

```python
g = 'mississippi'
d = defaultdict(int)
for k in g:
    ... d[k] += 1
...
d.items()
[('i', 4), ('p', 2), ('s', 4), ('m', 1)]
```

When a letter is first encountered, it is missing from the mapping, so the `default_factory` function calls `int()` to supply a default count of zero. The increment operation then builds up the count for each letter.

The function `int()` which always returns zero is just a special case of constant functions. A faster and more flexible way to create constant functions is to use `itertools.repeat()` which can supply any constant value (not just zero):

```python
def constant_factory(value):
    ... return itertools.repeat(value).next
g = defaultdict(constant_factory('<missing>'))
d.update(name='John', action='ran')
'%%%s %%(action)s to %%(object)s' % d
'John ran to <missing>'
```

Setting the `default_factory` to `set` makes the `defaultdict` useful for building a dictionary of sets:

```python
g = [(\'red\', 1), (\'blue\', 2), (\'red\', 3), (\'blue\', 4), (\'re
```
```python
>>> for k, v in s:
...     d[k].add(v)
...
>>> d.items()
[('blue', set([2, 4])), ('red', set([1, 3]))]
```
8.3.4. namedtuple() Factory Function for Tuples with Named Fields

Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code. They can be used wherever regular tuples are used, and they add the ability to access fields by name instead of position index.

```python
collections.namedtuple(typename, field_names[, verbose=False][, rename=False])
```

Returns a new tuple subclass named `typename`. The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being indexable and iterable. Instances of the subclass also have a helpful docstring (with `typename` and `field_names`) and a helpful `__repr__()` method which lists the tuple contents in a `name=value` format.

The `field_names` are a sequence of strings such as `['x', 'y']`. Alternatively, `field_names` can be a single string with each fieldname separated by whitespace and/or commas, for example `'x y'` or `'x, y'`.

Any valid Python identifier may be used for a fieldname except for names starting with an underscore. Valid identifiers consist of letters, digits, and underscores but do not start with a digit or underscore and cannot be a keyword such as `class`, `for`, `return`, `global`, `pass`, `print`, or `raise`.

If `rename` is true, invalid fieldnames are automatically replaced with positional names. For example, `['abc', 'def', 'ghi', 'abc']` is converted to `['abc', '_1', 'ghi', '_3']`, eliminating the keyword `def` and the duplicate fieldname `abc`.
If `verbose` is true, the class definition is printed just before being built.

Named tuple instances do not have per-instance dictionaries, so they are lightweight and require no more memory than regular tuples.

*New in version 2.6.*

*Changed in version 2.7:* added support for `rename`.

Example:

```python
globals().update(locals())

Point = namedtuple('Point', ['x', 'y'], verbose=True)

class Point(tuple):
    __slots__ = ()
    _fields = ('x', 'y')

    def __new__(cls, x, y):
        'Create new instance of Point(x, y)'
        return _tuple.__new__(cls, (x, y))

    @classmethod
    def _make(cls, iterable, new=tuple.__new__, len=len):
        'Make a new Point object from a sequence or iterable'
        result = new(cls, iterable)
        if len(result) != 2:
            raise TypeError('Expected 2 arguments, got %d' % len(result))
        return result

    def __repr__(self):
        'Return a nicely formatted representation string'
        return 'Point(x=%r, y=%r)' % self

    def _asdict(self):
        'Return a new OrderedDict which maps field names to their values'
        return OrderedDict(zip(self._fields, self))

    def _replace(_self, **kwds):
        'Return a new Point object replacing specified fields with new values'
```
result = _self._make(map(kwds.pop, ('x', 'y'), _self))
if kwds:
    raise ValueError('Got unexpected field names: %r' % kwds.keys())
return result

def __getnewargs__(self):
    'Return self as a plain tuple. Used by copy and pickle'
    return tuple(self)

__dict__ = _property(_asdict)

def __getstate__(self):
    'Exclude the OrderedDict from pickling'
    pass

x = _property(_itemgetter(0), doc='Alias for field number 0
y = _property(_itemgetter(1), doc='Alias for field number 1

>>> p = Point(11, y=22) # instantiate with positional or keyword arguments
>>> p[0] + p[1]         # indexable like the plain tuple (1, 22)
33
>>> x, y = p            # unpack like a regular tuple
>>> (11, 22)
>>> p.x + p.y           # fields also accessible by name
33
>>> p                   # readable __repr__ with a name=value style
Point(x=11, y=22)

Named tuples are especially useful for assigning field names to result tuples returned by the csv or sqlite3 modules:

```python
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees
print emp.name, emp.title

import sqlite3
conn = sqlite3.connect("/companydata")
cursor = conn.cursor()
```
In addition to the methods inherited from tuples, named tuples support three additional methods and one attribute. To prevent conflicts with field names, the method and attribute names start with an underscore.

**classmethod** `somenamedtuple._make(iterable)`
Class method that makes a new instance from an existing sequence or iterable.

```python
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

**somenamedtuple._asdict()**
Return a new `OrderedDict` which maps field names to their corresponding values:

```python
>>> p = Point(x=11, y=22)
>>> p._asdict()
OrderedDict([('x', 11), ('y', 22)])
```

*Changed in version 2.7:* Returns an `OrderedDict` instead of a regular `dict`.

**somenamedtuple._replace(kwars)**
Return a new instance of the named tuple replacing specified fields with new values:

```python
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)
```
somenamedtuple._fields

Tuple of strings listing the field names. Useful for introspection and for creating new named tuple types from existing named tuples.

```python
>>> p._fields  # view the field names
('x', 'y')
```

```python
>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
```

```python
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

To retrieve a field whose name is stored in a string, use the `getattr()` function:

```python
>>> getattr(p, 'x')
11
```

To convert a dictionary to a named tuple, use the double-star-operator (as described in Unpacking Argument Lists):

```python
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

Since a named tuple is a regular Python class, it is easy to add or change functionality with a subclass. Here is how to add a calculated field and a fixed-width print format:

```python
>>> class Point(namedtuple('Point', 'x y')):
...     __slots__ = ()
...     @property
...     def hypot(self):
...             return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...             return 'Point: x=%.3f y=%.3f hypot=%.3f' % (self.x, self.y, self.hypot)
```
... for p in Point(3, 4), Point(14, 5/7):
...     print p
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018

The subclass shown above sets __slots__ to an empty tuple. This helps keep memory requirements low by preventing the creation of instance dictionaries.

Subclassing is not useful for adding new, stored fields. Instead, simply create a new named tuple type from the _fields attribute:

```python
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

Default values can be implemented by using _replace() to customize a prototype instance:

```python
>>> Account = namedtuple('Account', 'owner balance transaction_count')
>>> default_account = Account('<owner name>', 0.0, 0)
>>> johns_account = default_account._replace(owner='John')
```

Enumerated constants can be implemented with named tuples, but it is simpler and more efficient to use a simple class declaration:

```python
>>> Status = namedtuple('Status', 'open pending closed')._make((0, 1, 2))
>>> class Status:
...     open, pending, closed = range(3)
```

See also: Named tuple recipe adapted for Python 2.4.
8.3.5. OrderedDict objects

Ordered dictionaries are just like regular dictionaries but they remember the order that items were inserted. When iterating over an ordered dictionary, the items are returned in the order their keys were first added.

```python
class collections.OrderedDict([items])
```

Return an instance of a dict subclass, supporting the usual `dict` methods. An `OrderedDict` is a dict that remembers the order that keys were first inserted. If a new entry overwrites an existing entry, the original insertion position is left unchanged. Deleting an entry and reinserting it will move it to the end.

*New in version 2.7.*

```python
OrderedDict.popitem(last=True)
```

The `popitem()` method for ordered dictionaries returns and removes a (key, value) pair. The pairs are returned in LIFO order if `last` is true or FIFO order if false.

In addition to the usual mapping methods, ordered dictionaries also support reverse iteration using `reversed()`.

Equality tests between `OrderedDict` objects are order-sensitive and are implemented as `list(od1.items())==list(od2.items())`. Equality tests between `OrderedDict` objects and other `Mapping` objects are order-insensitive like regular dictionaries. This allows `OrderedDict` objects to be substituted anywhere a regular dictionary is used.

The `OrderedDict` constructor and `update()` method both accept keyword arguments, but their order is lost because Python’s function
call semantics pass-in keyword arguments using a regular unordered dictionary.

See also: Equivalent OrderedDict recipe that runs on Python 2.4 or later.

8.3.5.1. OrderedDict Examples and Recipes

Since an ordered dictionary remembers its insertion order, it can be used in conjunction with sorting to make a sorted dictionary:

```
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```

The new sorted dictionaries maintain their sort order when entries are deleted. But when new keys are added, the keys are appended to the end and the sort is not maintained.

It is also straight-forward to create an ordered dictionary variant that remembers the order the keys were last inserted. If a new entry overwrites an existing entry, the original insertion position is changed and moved to the end:

```python
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'
```
def __setitem__(self, key, value):
    if key in self:
        del self[key]
    OrderedDict.__setitem__(self, key, value)

An ordered dictionary can be combined with the Counter class so that the counter remembers the order elements are first encountered:

class OrderedCounter(Counter, OrderedDict):
    'Counter that remembers the order elements are first encoun

    def __repr__(self):
        return '%s(%r)' % (self.__class__.__name__, OrderedDict(self)

    def __reduce__(self):
        return self.__class__, (OrderedDict(self),)
### 8.3.6. Collections Abstract Base Classes

The collections module offers the following ABCs:

<table>
<thead>
<tr>
<th>ABC</th>
<th>Inherits from</th>
<th>Abstract Methods</th>
<th>Mixin Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container</td>
<td></td>
<td><strong>contains</strong></td>
<td></td>
</tr>
<tr>
<td>Hashable</td>
<td></td>
<td><strong>hash</strong></td>
<td></td>
</tr>
<tr>
<td>Iterable</td>
<td></td>
<td><strong>iter</strong></td>
<td></td>
</tr>
<tr>
<td>Iterator</td>
<td>Iterable</td>
<td>next, <strong>iter</strong></td>
<td></td>
</tr>
<tr>
<td>Sized</td>
<td></td>
<td><strong>len</strong></td>
<td></td>
</tr>
<tr>
<td>Callable</td>
<td></td>
<td><strong>call</strong></td>
<td></td>
</tr>
<tr>
<td>Sequence</td>
<td>Sized, Iterable, Container</td>
<td><strong>getitem</strong>, <strong>len</strong></td>
<td><strong>contains</strong>, <strong>iter</strong>, <strong>reversed</strong>, index, and count</td>
</tr>
<tr>
<td>MutableSequence</td>
<td>Sequence</td>
<td><strong>getitem</strong>, <strong>setitem</strong>, <strong>delitem</strong>, <strong>len</strong>, insert</td>
<td>Inherited Sequence methods and append, reverse, extend, pop, remove, and <strong>iadd</strong></td>
</tr>
<tr>
<td>Set</td>
<td>Sized, Iterable, Container</td>
<td><strong>contains</strong>, <strong>iter</strong>, <strong>len</strong></td>
<td><strong>le</strong>, <strong>lt</strong>, <strong>eq</strong>, <strong>ne</strong>, <strong>gt</strong>, <strong>ge</strong>, <strong>and</strong>, <strong>or</strong>, <strong>sub</strong>, <strong>xor</strong>, and isdisjoint</td>
</tr>
<tr>
<td>MutableSet</td>
<td>Set</td>
<td><strong>contains</strong>, <strong>iter</strong>, <strong>len</strong>, add, discard</td>
<td>Inherited Set methods and clear, pop, remove, <strong>ior</strong>, <strong>iand</strong>, <strong>ixor</strong>, and <strong>isub</strong></td>
</tr>
<tr>
<td>Mapping</td>
<td>Sized, Iterable,</td>
<td><strong>getitem</strong>, <strong>iter</strong></td>
<td><strong>contains</strong>, keys, items, values, get,</td>
</tr>
<tr>
<td>Class</td>
<td>Inheritance</td>
<td>Methods</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------</td>
<td>----------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><code>collections.Container</code></td>
<td></td>
<td><code>__len__</code>, <code>__eq__</code>, and <code>__ne__</code></td>
<td></td>
</tr>
<tr>
<td><code>collections.MutableMapping</code></td>
<td><code>Mapping</code></td>
<td><code>__getitem__</code>, <code>__setitem__</code>, <code>__delitem__</code>, <code>__iter__</code>, <code>__len__</code></td>
<td></td>
</tr>
<tr>
<td><code>collections.Mapping</code></td>
<td></td>
<td>Inherited Mapping methods and <code>pop</code>, <code>popitem</code>, <code>clear</code>, <code>update</code>, and <code>setdefault</code></td>
<td></td>
</tr>
<tr>
<td><code>collections.MappingView</code></td>
<td><code>Sized</code></td>
<td><code>__len__</code></td>
<td></td>
</tr>
<tr>
<td><code>collections.ItemsView</code></td>
<td><code>Set</code></td>
<td><code>__contains__</code>, <code>__iter__</code></td>
<td></td>
</tr>
<tr>
<td><code>collections.KeysView</code></td>
<td><code>Set</code></td>
<td><code>__contains__</code>, <code>__iter__</code></td>
<td></td>
</tr>
<tr>
<td><code>collections.ValuesView</code></td>
<td><code>MappingView</code></td>
<td><code>__contains__</code>, <code>__iter__</code></td>
<td></td>
</tr>
</tbody>
</table>

- **class collections.Container**
  - Inheritance:
  - `collections.mutablemapping`
  - Methods: `__len__`, `__eq__`, `__ne__`

- **class collections.MutableMapping**
  - Inheritance:
  - `collections.mapping`
  - Methods: `__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__`

- **class collections.Mapping**
  - Inheritance:
  - `collections.mapping`
  - Methods: Inherited Mapping methods and `pop`, `popitem`, `clear`, `update`, and `setdefault`

- **class collections.MappingView**
  - Inheritance:
  - `collections.sized`
  - Methods: `__len__`

- **class collections.ItemsView**
  - Inheritance:
  - `collections.mappingview`, `collections.set`
  - Methods: `__contains__`, `__iter__`

- **class collections.KeysView**
  - Inheritance:
  - `collections.mappingview`, `collections.set`
  - Methods: `__contains__`, `__iter__`

- **class collections.ValuesView**
  - Inheritance:
  - `collections.mappingview`
  - Methods: `__contains__`, `__iter__`

**class collections.Hashable**
- ABCs for classes that provide respectively the methods `__contains__()`, `__hash__()`, `__len__()`, and `__call__()`.

**class collections.Sized**
- ABC for classes that provide the `__iter__()` method. See also the definition of `iterable`.

**class collections.Callable**
- ABC for classes that provide the `__iter__()` and `next()` methods. See also the definition of `iterator`.

**class collections.Iterable**
- ABC for classes that provide the `__iter__()` method. See also the definition of `iterable`.

**class collections.Sequence**
- ABCs for read-only and mutable sequences.

**class collections.Set**
- ABC for classes that provide the `__iter__()` and `next()` methods. See also the definition of `iterator`.

**class collections.MutableSet**
- ABCs for read-only and mutable sets.
**class** collections.**Mapping**  
**class** collections.**MutableMapping**  
  ABCs for read-only and mutable mappings.

**class** collections.**MappingView**  
**class** collections.**ItemsView**  
**class** collections.**KeysView**  
**class** collections.**ValuesView**  
  ABCs for mapping, items, keys, and values views.

These ABCs allow us to ask classes or instances if they provide particular functionality, for example:

```python
size = None
if isinstance(myvar, collections.Sized):
    size = len(myvar)
```

Several of the ABCs are also useful as mixins that make it easier to develop classes supporting container APIs. For example, to write a class supporting the full set API, it only necessary to supply the three underlying abstract methods: `__contains__()`, `__iter__()`, and `__len__()`. The ABC supplies the remaining methods such as `__and__()` and `isdisjoint()`

```python
class ListBasedSet(collections.Set):
    ''' Alternate set implementation favoring space over speed and not requiring the set elements to be hashable. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)

    def __iter__(self):
        return iter(self.elements)

    def __contains__(self, value):
        return value in self.elements
```
```python
def __len__(self):
    return len(self.elements)
```

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2  # The __and__() method is supported

Notes on using `Set` and `MutableSet` as a mixin:

1. Since some set operations create new sets, the default mixin methods need a way to create new instances from an iterable. The class constructor is assumed to have a signature in the form `ClassName(iterable)`. That assumption is factored-out to an internal classmethod called `_from_iterable()` which calls `cls(iterable)` to produce a new set. If the `set` mixin is being used in a class with a different constructor signature, you will need to override `_from_iterable()` with a classmethod that can construct new instances from an iterable argument.

2. To override the comparisons (presumably for speed, as the semantics are fixed), redefine `__le__()` and `__ge__()`, then the other operations will automatically follow suit.

3. The `Set` mixin provides a `_hash()` method to compute a hash value for the set; however, `__hash__()` is not defined because not all sets are hashable or immutable. To add set hashability using mixins, inherit from both `Set()` and `Hashable()`, then define `__hash__ = Set._hash`.

**See also:**

- `OrderedSet recipe` for an example built on `MutableSet`.
- For more about ABCs, see the `abc` module and PEP 3119.
8.4. heapq — Heap queue algorithm

*New in version 2.3.*

**Source code:** Lib/heapq.py

This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

Heaps are binary trees for which every parent node has a value less than or equal to any of its children. This implementation uses arrays for which $heap[k] \leq heap[2*k+1]$ and $heap[k] \leq heap[2*k+2]$ for all $k$, counting elements from zero. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that its smallest element is always the root, $heap[0]$.

The API below differs from textbook heap algorithms in two aspects: (a) We use zero-based indexing. This makes the relationship between the index for a node and the indexes for its children slightly less obvious, but is more suitable since Python uses zero-based indexing. (b) Our pop method returns the smallest item, not the largest (called a “min heap” in textbooks; a “max heap” is more common in texts because of its suitability for in-place sorting).

These two make it possible to view the heap as a regular Python list without surprises: $heap[0]$ is the smallest item, and $heap.sort()$ maintains the heap invariant!

To create a heap, use a list initialized to [], or you can transform a populated list into a heap via function `heapify()`.

The following functions are provided:
heapq. **heappush**(heap, item)

Push the value *item* onto the *heap*, maintaining the heap invariant.

heapq. **heappop**(heap)

Pop and return the smallest item from the *heap*, maintaining the heap invariant. If the heap is empty, *IndexError* is raised. To access the smallest item without popping it, use *heap[0]*.

heapq. **heappushpop**(heap, item)

Push *item* on the heap, then pop and return the smallest item from the *heap*. The combined action runs more efficiently than **heappush()** followed by a separate call to **heappop()**.

*New in version 2.6.*

heapq. **heapify**(x)

Transform list *x* into a heap, in-place, in linear time.

heapq. **heapreplace**(heap, item)

Pop and return the smallest item from the *heap*, and also push the new *item*. The heap size doesn’t change. If the heap is empty, *IndexError* is raised.

This one step operation is more efficient than a **heappop()** followed by **heappush()** and can be more appropriate when using a fixed-size heap. The pop/push combination always returns an element from the heap and replaces it with *item*.

The value returned may be larger than the *item* added. If that isn’t desired, consider using **heappushpop()** instead. Its push/pop combination returns the smaller of the two values, leaving the larger value on the heap.

The module also offers three general purpose functions based on
heaps.

`heapq.merge(*iterables)`

Merge multiple sorted inputs into a single sorted output (for example, merge timestamped entries from multiple log files). Returns an `iterator` over the sorted values.

Similar to `sorted(itertools.chain(*iterables))` but returns an iterable, does not pull the data into memory all at once, and assumes that each of the input streams is already sorted (smallest to largest).

*New in version 2.6.*

`heapq.nlargest(n, iterable[, key])`

Return a list with the `n` largest elements from the dataset defined by `iterable`. `key`, if provided, specifies a function of one argument that is used to extract a comparison key from each element in the `iterable`: `key=str.lower`  
Equivalent to: `sorted(iterable, key=key, reverse=True)[:n]`

*New in version 2.4.*

*Changed in version 2.5:* Added the optional `key` argument.

`heapq.nsmallest(n, iterable[, key])`

Return a list with the `n` smallest elements from the dataset defined by `iterable`. `key`, if provided, specifies a function of one argument that is used to extract a comparison key from each element in the `iterable`: `key=str.lower`  
Equivalent to: `sorted(iterable, key=key)[:n]`

*New in version 2.4.*

*Changed in version 2.5:* Added the optional `key` argument.
The latter two functions perform best for smaller values of $n$. For larger values, it is more efficient to use the `sorted()` function. Also, when $n=1$, it is more efficient to use the built-in `min()` and `max()` functions. If repeated usage of these functions is required, consider turning the iterable into an actual heap.
A **heapsort** can be implemented by pushing all values onto a heap and then popping off the smallest values one at a time:

```python
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This is similar to `sorted(iterable)`, but unlike `sorted()`, this implementation is not stable.

Heap elements can be tuples. This is useful for assigning comparison values (such as task priorities) alongside the main record being tracked:

```python
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```
8.4.2. Priority Queue Implementation Notes

A priority queue is common use for a heap, and it presents several implementation challenges:

- Sort stability: how do you get two tasks with equal priorities to be returned in the order they were originally added?
- In the future with Python 3, tuple comparison breaks for (priority, task) pairs if the priorities are equal and the tasks do not have a default comparison order.
- If the priority of a task changes, how do you move it to a new position in the heap?
- Or if a pending task needs to be deleted, how do you find it and remove it from the queue?

A solution to the first two challenges is to store entries as 3-element list including the priority, an entry count, and the task. The entry count serves as a tie-breaker so that two tasks with the same priority are returned in the order they were added. And since no two entry counts are the same, the tuple comparison will never attempt to directly compare two tasks.

The remaining challenges revolve around finding a pending task and making changes to its priority or removing it entirely. Finding a task can be done with a dictionary pointing to an entry in the queue.

Removing the entry or changing its priority is more difficult because it would break the heap structure invariants. So, a possible solution is to mark the existing entry as removed and add a new entry with the revised priority:

```python
pq = []
entry_finder = {}
REMOVED = '<removed-task>'
counter = itertools.count()  # list of entries arranged in a
data # mapping of tasks to entries # placeholder for a removed task # unique sequence count
```
def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    'Mark an existing task as REMOVED. Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    'Remove and return the lowest priority task. Raise KeyError'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')
8.4.3. Theory

Heaps are arrays for which \( a[k] \leq a[2k+1] \) and \( a[k] \leq a[2k+2] \) for all \( k \), counting elements from 0. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that \( a[0] \) is always its smallest element.

The strange invariant above is meant to be an efficient memory representation for a tournament. The numbers below are \( k \), not \( a[k] \):

\[
\begin{array}{cccccccccccccc}
0 & 1 & 2 \\
3 & 4 & 5 & 6 \\
7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\end{array}
\]

In the tree above, each cell \( k \) is topping \( 2k+1 \) and \( 2k+2 \). In a usual binary tournament we see in sports, each cell is the winner over the two cells it tops, and we can trace the winner down the tree to see all opponents s/he had. However, in many computer applications of such tournaments, we do not need to trace the history of a winner. To be more memory efficient, when a winner is promoted, we try to replace it by something else at a lower level, and the rule becomes that a cell and the two cells it tops contain three different items, but the top cell “wins” over the two topped cells.

If this heap invariant is protected at all time, index 0 is clearly the overall winner. The simplest algorithmic way to remove it and find the “next” winner is to move some loser (let’s say cell 30 in the diagram above) into the 0 position, and then percolate this new 0 down the tree, exchanging values, until the invariant is re-established. This is
clearly logarithmic on the total number of items in the tree. By iterating over all items, you get an $O(n \log n)$ sort.

A nice feature of this sort is that you can efficiently insert new items while the sort is going on, provided that the inserted items are not "better" than the last 0'th element you extracted. This is especially useful in simulation contexts, where the tree holds all incoming events, and the “win” condition means the smallest scheduled time. When an event schedules other events for execution, they are scheduled into the future, so they can easily go into the heap. So, a heap is a good structure for implementing schedulers (this is what I used for my MIDI sequencer :-).

Various structures for implementing schedulers have been extensively studied, and heaps are good for this, as they are reasonably speedy, the speed is almost constant, and the worst case is not much different than the average case. However, there are other representations which are more efficient overall, yet the worst cases might be terrible.

Heaps are also very useful in big disk sorts. You most probably all know that a big sort implies producing “runs” (which are pre-sorted sequences, whose size is usually related to the amount of CPU memory), followed by a merging passes for these runs, which merging is often very cleverly organised [1]. It is very important that the initial sort produces the longest runs possible. Tournaments are a good way to achieve that. If, using all the memory available to hold a tournament, you replace and percolate items that happen to fit in the current run, you'll produce runs which are twice the size of the memory for random input, and much better for input fuzzily ordered.

Moreover, if you output the 0'th item on disk and get an input which may not fit in the current tournament (because the value “wins” over the last output value), it cannot fit in the heap, so the size of the heap decreases. The freed memory could be cleverly reused immediately
for progressively building a second heap, which grows at exactly the same rate the first heap is melting. When the first heap completely vanishes, you switch heaps and start a new run. Clever and quite effective!

In a word, heaps are useful memory structures to know. I use them in a few applications, and I think it is good to keep a ‘heap’ module around. :-)

Footnotes

The disk balancing algorithms which are current, nowadays, are more annoying than clever, and this is a consequence of the seeking capabilities of the disks. On devices which cannot seek, like big tape drives, the story was quite different, and one had to be very clever to ensure (far in advance) that each tape movement will be the most effective possible (that is, will best participate at “progressing” the merge). Some tapes were even able to read backwards, and this was also used to avoid the rewinding time. Believe me, real good tape sorts were quite spectacular to watch! From all times, sorting has always been a Great Art! :-)
New in version 2.1.

Source code: Lib/bisect.py

This module provides support for maintaining a list in sorted order without having to sort the list after each insertion. For long lists of items with expensive comparison operations, this can be an improvement over the more common approach. The module is called bisect because it uses a basic bisection algorithm to do its work. The source code may be most useful as a working example of the algorithm (the boundary conditions are already right!).

The following functions are provided:

```python
bisect.bisect_left(a, x, lo=0, hi=len(a))
```

Locate the insertion point for \( x \) in \( a \) to maintain sorted order. The parameters \( lo \) and \( hi \) may be used to specify a subset of the list which should be considered; by default the entire list is used. If \( x \) is already present in \( a \), the insertion point will be before (to the left of) any existing entries. The return value is suitable for use as the first parameter to `list.insert()` assuming that \( a \) is already sorted.

The returned insertion point \( i \) partitions the array \( a \) into two halves so that \[ \text{all}(\text{val} < x \text{ for } \text{val} \text{ in } a[lo:i]) \] for the left side and \[ \text{all} \left( \text{val} \geq x \text{ for } \text{val} \text{ in } a[i:hi] \right) \] for the right side.

```python
bisect.bisect_right(a, x, lo=0, hi=len(a))
bisect.bisect(a, x, lo=0, hi=len(a))
```
Similar to `bisect_left()`, but returns an insertion point which comes after (to the right of) any existing entries of \( x \) in \( a \).

The returned insertion point \( i \) partitions the array \( a \) into two halves so that \( \text{all}(\text{val} <= \text{x} \text{ for } \text{val in } a[\text{lo}:\text{i}]) \) for the left side and \( \text{all}(\text{val} > \text{x} \text{ for } \text{val in } a[\text{i}:\text{hi}]) \) for the right side.

\[
\text{bisect.\texttt{insort_left}}(a, x, \text{lo}=0, \text{hi}=\text{len}(a))
\]
Insert \( x \) in \( a \) in sorted order. This is equivalent to
\[
a.\text{insert}(\text{bisect.bisect_left}(a, x, \text{lo}, \text{hi}), x)
\]
assuming that \( a \) is already sorted. Keep in mind that the \( O(\log n) \) search is dominated by the slow \( O(n) \) insertion step.

\[
\text{bisect.\texttt{insort_right}}(a, x, \text{lo}=0, \text{hi}=\text{len}(a))
\]
\[
\text{bisect.\texttt{insort}}(a, x, \text{lo}=0, \text{hi}=\text{len}(a))
\]
Similar to `\texttt{insort_left()}`, but inserting \( x \) in \( a \) after any existing entries of \( x \).

See also: SortedCollection recipe that uses bisect to build a full-featured collection class with straight-forward search methods and support for a key-function. The keys are precomputed to save unnecessary calls to the key function during searches.
8.5.1. Searching Sorted Lists

The above \texttt{bisect()} functions are useful for finding insertion points but can be tricky or awkward to use for common searching tasks. The following five functions show how to transform them into the standard lookups for sorted lists:

\begin{verbatim}
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
\end{verbatim}
8.5.2. Other Examples

The \texttt{bisect()} function can be useful for numeric table lookups. This example uses \texttt{bisect()} to look up a letter grade for an exam score (say) based on a set of ordered numeric breakpoints: 90 and up is an ‘A’, 80 to 89 is a ‘B’, and so on:

```python
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
    i = bisect(breakpoints, score)
    return grades[i]

>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

Unlike the \texttt{sorted()} function, it does not make sense for the \texttt{bisect()} functions to have \textit{key} or \textit{reversed} arguments because that would lead to an inefficient design (successive calls to bisect functions would not “remember” all of the previous key lookups).

Instead, it is better to search a list of precomputed keys to find the index of the record in question:

```python
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=\texttt{lambda} r: r[1])
>>> keys = [r[1] \textbf{for} r \textbf{in} data]       \# precomputed list of keys
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```
This module defines an object type which can compactly represent an array of basic values: characters, integers, floating point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a type code, which is a single character. The following type codes are defined:

<table>
<thead>
<tr>
<th>Type code</th>
<th>C Type</th>
<th>Python Type</th>
<th>Minimum size in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>'c'</td>
<td>char</td>
<td>character</td>
<td>1</td>
</tr>
<tr>
<td>'b'</td>
<td>signed char</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>'B'</td>
<td>unsigned char</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>'u'</td>
<td>Py_UNICODE</td>
<td>Unicode character</td>
<td>2 (see note)</td>
</tr>
<tr>
<td>'h'</td>
<td>signed short</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>'H'</td>
<td>unsigned short</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>'i'</td>
<td>signed int</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>'I'</td>
<td>unsigned int</td>
<td>long</td>
<td>2</td>
</tr>
<tr>
<td>'l'</td>
<td>signed long</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>'L'</td>
<td>unsigned long</td>
<td>long</td>
<td>4</td>
</tr>
<tr>
<td>'f'</td>
<td>float</td>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>'d'</td>
<td>double</td>
<td>float</td>
<td>8</td>
</tr>
</tbody>
</table>

**Note:** The 'u' type code corresponds to Python's unicode character. On narrow Unicode builds this is 2-bytes, on wide builds this is 4-bytes.

The actual representation of values is determined by the machine
architecture (strictly speaking, by the C implementation). The actual size can be accessed through the \texttt{itemsize} attribute. The values stored for \texttt{'L'} and \texttt{'I'} items will be represented as Python long integers when retrieved, because Python's plain integer type cannot represent the full range of C's unsigned (long) integers.

The module defines the following type:

\begin{verbatim}
class array.array(typecode[, initializer])
A new array whose items are restricted by \texttt{typecode}, and initialized from the optional \texttt{initializer} value, which must be a list, string, or iterable over elements of the appropriate type.

\textit{Changed in version 2.4}: Formerly, only lists or strings were accepted.

If given a list or string, the initializer is passed to the new array's \texttt{fromlist()}, \texttt{fromstring()}, or \texttt{fromunicode()} method (see below) to add initial items to the array. Otherwise, the iterable initializer is passed to the \texttt{extend()} method.
\end{verbatim}

\texttt{array.ArrayType}

Obsolete alias for \texttt{array}.

Array objects support the ordinary sequence operations of indexing, slicing, concatenation, and multiplication. When using slice assignment, the assigned value must be an array object with the same type code; in all other cases, \texttt{TypeError} is raised. Array objects also implement the buffer interface, and may be used wherever buffer objects are supported.

The following data items and methods are also supported:

\texttt{array.typecode}

The typecode character used to create the array.
array.**itemsize**

The length in bytes of one array item in the internal representation.

array.**append**(x)

Append a new item with value x to the end of the array.

array.**buffer_info()**

Return a tuple \((\text{address}, \text{length})\) giving the current memory address and the length in elements of the buffer used to hold array’s contents. The size of the memory buffer in bytes can be computed as `array.buffer_info()[1] * array.itemsize`. This is occasionally useful when working with low-level (and inherently unsafe) I/O interfaces that require memory addresses, such as certain `ioctl()` operations. The returned numbers are valid as long as the array exists and no length-changing operations are applied to it.

**Note:** When using array objects from code written in C or C++ (the only way to effectively make use of this information), it makes more sense to use the buffer interface supported by array objects. This method is maintained for backward compatibility and should be avoided in new code. The buffer interface is documented in Buffers and Memoryview Objects.

array.**byteswap()**

“Byteswap” all items of the array. This is only supported for values which are 1, 2, 4, or 8 bytes in size; for other types of values, `RuntimeError` is raised. It is useful when reading data from a file written on a machine with a different byte order.

array.**count**(x)

Return the number of occurrences of x in the array.
array.extend(iterable)

Append items from iterable to the end of the array. If iterable is another array, it must have exactly the same type code; if not, TypeError will be raised. If iterable is not an array, it must be iterable and its elements must be the right type to be appended to the array.

*Changed in version 2.4:* Formerly, the argument could only be another array.

array.fromfile(f, n)

Read n items (as machine values) from the file object f and append them to the end of the array. If less than n items are available, EOFError is raised, but the items that were available are still inserted into the array. f must be a real built-in file object; something else with a read() method won’t do.

array.fromlist(list)

Append items from the list. This is equivalent to for x in list: a.append(x) except that if there is a type error, the array is unchanged.

array.fromstring(s)

Appends items from the string, interpreting the string as an array of machine values (as if it had been read from a file using the fromfile() method).

array.fromunicode(s)

Extends this array with data from the given unicode string. The array must be a type 'u' array; otherwise a ValueError is raised. Use array.fromstring(unicodestring.encode(enc)) to append Unicode data to an array of some other type.

array.index(x)
Return the smallest \( i \) such that \( i \) is the index of the first occurrence of \( x \) in the array.

\[ \text{array}. \text{insert}(i, x) \]
Insert a new item with value \( x \) in the array before position \( i \). Negative values are treated as being relative to the end of the array.

\[ \text{array}. \text{pop}([i]) \]
Removes the item with the index \( i \) from the array and returns it. The optional argument defaults to -1, so that by default the last item is removed and returned.

\[ \text{array}. \text{read}(f, n) \]
Deprecated since version 1.5.1: Use the fromfile() method.
Read \( n \) items (as machine values) from the file object \( f \) and append them to the end of the array. If less than \( n \) items are available, \text{EOFError} is raised, but the items that were available are still inserted into the array. \( f \) must be a real built-in file object; something else with a read() method won’t do.

\[ \text{array}. \text{remove}(x) \]
Remove the first occurrence of \( x \) from the array.

\[ \text{array}. \text{reverse}() \]
Reverse the order of the items in the array.

\[ \text{array}. \text{tofile}(f) \]
Write all items (as machine values) to the file object \( f \).

\[ \text{array}. \text{tolist}() \]
Convert the array to an ordinary list with the same items.

\[ \text{array}. \text{tostring}() \]
Convert the array to an array of machine values and return the string representation (the same sequence of bytes that would be written to a file by the `tofile()` method.)

`array.tounicode()`

Convert the array to a unicode string. The array must be a type `'u'` array; otherwise a `ValueError` is raised. Use `array.tostring().decode(enc)` to obtain a unicode string from an array of some other type.

`array.write(f)`

*Deprecated since version 1.5.1: Use the `tofile()` method.*

Write all items (as machine values) to the file object `f`.

When an array object is printed or converted to a string, it is represented as `array(typecode, initializer)`. The `initializer` is omitted if the array is empty, otherwise it is a string if the `typecode` is `'c'`, otherwise it is a list of numbers. The string is guaranteed to be able to be converted back to an array with the same type and value using `eval()`, so long as the `array` class has been imported using `from array import array`. Examples:

```python
array('l')
array('c', 'hello world')
array('u', u'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

**See also:**

- **Module `struct`**
  - Packing and unpacking of heterogeneous binary data.
- **Module `xdrlib`**
  - Packing and unpacking of External Data Representation (XDR)
data as used in some remote procedure call systems.

The Numerical Python Documentation
The Numeric Python extension (NumPy) defines another array type; see http://www.numpy.org/ for further information about Numerical Python.
8.7. sets — Unordered collections of unique elements

New in version 2.3.

Deprecated since version 2.6: The built-in `set/frozenset` types replace this module.

The `sets` module provides classes for constructing and manipulating unordered collections of unique elements. Common uses include membership testing, removing duplicates from a sequence, and computing standard math operations on sets such as intersection, union, difference, and symmetric difference.

Like other collections, sets support `x in set`, `len(set)`, and `for x in set`. Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

Most set applications use the `Set` class which provides every set method except for `__hash__()`. For advanced applications requiring a hash method, the `ImmutableSet` class adds a `__hash__()` method but omits methods which alter the contents of the set. Both `Set` and `ImmutableSet` derive from `BaseSet`, an abstract class useful for determining whether something is a set: `isinstance(obj, BaseSet)`.

The set classes are implemented using dictionaries. Accordingly, the requirements for set elements are the same as those for dictionary keys; namely, that the element defines both `__eq__()` and `__hash__()`.

As a result, sets cannot contain mutable elements such as lists or dictionaries. However, they can contain immutable
collections such as tuples or instances of `ImmutableSet`. For convenience in implementing sets of sets, inner sets are automatically converted to immutable form, for example,

```
Set([Set(['dog'])])
```

is transformed to

```
Set([ImmutableSet(['dog'])])
```

**class sets.Set([iterable])**

Constructs a new empty `set` object. If the optional `iterable` parameter is supplied, updates the set with elements obtained from iteration. All of the elements in `iterable` should be immutable or be transformable to an immutable using the protocol described in section [Protocol for automatic conversion to immutable](#).

**class sets.ImmutableSet([iterable])**

Constructs a new empty `ImmutableSet` object. If the optional `iterable` parameter is supplied, updates the set with elements obtained from iteration. All of the elements in `iterable` should be immutable or be transformable to an immutable using the protocol described in section [Protocol for automatic conversion to immutable](#).

Because `ImmutableSet` objects provide a `__hash__()` method, they can be used as set elements or as dictionary keys. `ImmutableSet` objects do not have methods for adding or removing elements, so all of the elements must be known when the constructor is called.
8.7.1. Set Objects

Instances of `Set` and `ImmutableSet` both provide the following operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Equivalent</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(s)</code></td>
<td></td>
<td>number of elements in set <code>s</code> (cardinality)</td>
</tr>
<tr>
<td><code>x in s</code></td>
<td></td>
<td>test <code>x</code> for membership in <code>s</code></td>
</tr>
<tr>
<td><code>x not in s</code></td>
<td></td>
<td>test <code>x</code> for non-membership in <code>s</code></td>
</tr>
<tr>
<td><code>s.issubset(t)</code></td>
<td><code>s &lt;= t</code></td>
<td>test whether every element in <code>s</code> is in <code>t</code></td>
</tr>
<tr>
<td><code>s.issuperset(t)</code></td>
<td><code>s &gt;= t</code></td>
<td>test whether every element in <code>t</code> is in <code>s</code></td>
</tr>
<tr>
<td><code>s.union(t)</code></td>
<td>`s</td>
<td>t`</td>
</tr>
<tr>
<td><code>s.intersection(t)</code></td>
<td><code>s &amp; t</code></td>
<td>new set with elements common to <code>s</code> and <code>t</code></td>
</tr>
<tr>
<td><code>s.difference(t)</code></td>
<td><code>s - t</code></td>
<td>new set with elements in <code>s</code> but not in <code>t</code></td>
</tr>
<tr>
<td><code>s.symmetric_difference(t)</code></td>
<td><code>s ^ t</code></td>
<td>new set with elements in either <code>s</code> or <code>t</code> but not both</td>
</tr>
<tr>
<td><code>s.copy()</code></td>
<td></td>
<td>new set with a shallow copy of <code>s</code></td>
</tr>
</tbody>
</table>

Note, the non-operator versions of `union()`, `intersection()`, `difference()`, and `symmetric_difference()` will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes error-prone constructions like `Set('abc') & 'cbs'` in favor of the more readable `Set('abc').intersection('cbs')`. 
Changed in version 2.3.1: Formerly all arguments were required to be sets.

In addition, both Set and ImmutableSet support set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other). A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal). A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

The subset and equality comparisons do not generalize to a complete ordering function. For example, any two disjoint sets are not equal and are not subsets of each other, so all of the following return False: `a<b, a==b, or a>b`. Accordingly, sets do not implement the `__cmp__() method.

Since sets only define partial ordering (subset relationships), the output of the `list.sort()` method is undefined for lists of sets.

The following table lists operations available in ImmutableSet but not found in Set:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash(s)</td>
<td>returns a hash value for s</td>
</tr>
</tbody>
</table>

The following table lists operations available in Set but not found in ImmutableSet:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Equivalent</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.update(t)</td>
<td>s</td>
<td>= t</td>
</tr>
<tr>
<td>s.intersection_update(t)</td>
<td>s &amp; t</td>
<td>return set s keeping only</td>
</tr>
</tbody>
</table>
elements also found in \( t \)

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s.difference_update(t)</code></td>
<td><code>s -= t</code></td>
<td>return set ( s ) after removing elements found in ( t )</td>
</tr>
<tr>
<td><code>s.symmetric_difference_update(t)</code></td>
<td><code>s ^= t</code></td>
<td>return set ( s ) with elements from ( s ) or ( t ) but not both</td>
</tr>
<tr>
<td><code>s.add(x)</code></td>
<td></td>
<td>add element ( x ) to set ( s )</td>
</tr>
<tr>
<td><code>s.remove(x)</code></td>
<td></td>
<td>remove ( x ) from set ( s ); raises <code>KeyError</code> if not present</td>
</tr>
<tr>
<td><code>s.discard(x)</code></td>
<td></td>
<td>removes ( x ) from set ( s ) if present</td>
</tr>
<tr>
<td><code>s.pop()</code></td>
<td></td>
<td>remove and return an arbitrary element from ( s ); raises <code>KeyError</code> if empty</td>
</tr>
<tr>
<td><code>s.clear()</code></td>
<td></td>
<td>remove all elements from set ( s )</td>
</tr>
</tbody>
</table>

Note, the non-operator versions of `update()`, `intersection_update()`, `difference_update()`, and `symmetric_difference_update()` will accept any iterable as an argument.

*Changed in version 2.3.1*: Formerly all arguments were required to be sets.

Also note, the module also includes a `union_update()` method which is an alias for `update()`. The method is included for backwards compatibility. Programmers should prefer the `update()` method.
because it is supported by the built-in \texttt{set()} and \texttt{frozenset()} types.
8.7.2. Example

```python
>>> from sets import Set
>>> engineers = Set(['John', 'Jane', 'Jack', 'Janice'])
>>> programmers = Set(['Jack', 'Sam', 'Susan', 'Janice'])
>>> managers = Set(['Jane', 'Jack', 'Susan', 'Zack'])
>>> employees = engineers | programmers | managers  #
>>> engineering_management = engineers & managers  #
>>> fulltime_management = managers - engineers - programmers  #
>>> engineers.add('Marvin')  #
>>> print engineers
Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])
>>> employees.issuperset(engineers)  # superset test
False
>>> employees.update(engineers)  # update from another set
>>> employees.issuperset(engineers)
True
>>> for group in [engineers, programmers, managers, employees]:
...     group.discard('Susan')  # unconditionally remove
...     print group
...
Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])
Set(['Janice', 'Jack', 'Sam'])
Set(['Jane', 'Zack', 'Jack'])
Set(['Jack', 'Sam', 'Jane', 'Marvin', 'Janice', 'John', 'Zack'])
```
8.7.3. Protocol for automatic conversion to immutable

Sets can only contain immutable elements. For convenience, mutable `Set` objects are automatically copied to an `ImmutableSet` before being added as a set element.

The mechanism is to always add a `hashable` element, or if it is not hashable, the element is checked to see if it has an `__as_immutable__()` method which returns an immutable equivalent.

Since `Set` objects have a `__as_immutable__()` method returning an instance of `ImmutableSet`, it is possible to construct sets of sets.

A similar mechanism is needed by the `__contains__()` and `remove()` methods which need to hash an element to check for membership in a set. Those methods check an element for hashability and, if not, check for a `__as_temporarily_immutable__()` method which returns the element wrapped by a class that provides temporary methods for `__hash__()`, `__eq__()`, and `__ne__()`.

The alternate mechanism spares the need to build a separate copy of the original mutable object.

`Set` objects implement the `__as_temporarily_immutable__()` method which returns the `Set` object wrapped by a new class `_TemporarilyImmutableSet`.

The two mechanisms for adding hashability are normally invisible to the user; however, a conflict can arise in a multi-threaded environment where one thread is updating a set while another has temporarily wrapped it in `_TemporarilyImmutableSet`. In other words,
sets of mutable sets are not thread-safe.
The built-in `set` and `frozenset` types were designed based on lessons learned from the `sets` module. The key differences are:

- **Set** and **ImmutableSet** were renamed to **set** and **frozenset**.
- There is no equivalent to **BaseSet**. Instead, use `isinstance(x, (set, frozenset))`.
- The hash algorithm for the built-ins performs significantly better (fewer collisions) for most datasets.
- The built-in versions have more space efficient pickles.
- The built-in versions do not have a **union_update()** method. Instead, use the **update()** method which is equivalent.
- The built-in versions do not have a **_repr(sorted=True)** method. Instead, use the built-in **repr()** and **sorted()** functions: `repr(sorted(s))`.
- The built-in version does not have a protocol for automatic conversion to immutable. Many found this feature to be confusing and no one in the community reported having found real uses for it.
The `sched` module defines a class which implements a general purpose event scheduler:

```python
class sched.sched(scheduler(timefunc, delayfunc))
```

The `scheduler` class defines a generic interface to scheduling events. It needs two functions to actually deal with the “outside world” — `timefunc` should be callable without arguments, and return a number (the “time”, in any units whatsoever). The `delayfunc` function should be callable with one argument, compatible with the output of `timefunc`, and should delay that many time units. `delayfunc` will also be called with the argument 0 after each event is run to allow other threads an opportunity to run in multi-threaded applications.

Example:

```python
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(): print "From print_time", time.time()
...
>>> def print_some_times():
...    print time.time()
...    s.enter(5, 1, print_time, ()
...    s.enter(10, 1, print_time, ()
...    s.run()
...
>>> print_some_times()
930343690.257
From print_time 930343695.274
From print_time 930343700.273
930343700.276
```
In multi-threaded environments, the `scheduler` class has limitations with respect to thread-safety, inability to insert a new task before the one currently pending in a running scheduler, and holding up the main thread until the event queue is empty. Instead, the preferred approach is to use the `threading.Timer` class instead.

Example:

```python
>>> import time
>>> from threading import Timer
>>> def print_time():
...     print "From print_time", time.time()
...
>>> def print_some_times():
...     print time.time()
...     Timer(5, print_time, ()).start()
...     Timer(10, print_time, ()).start()
...     time.sleep(11)  # sleep while time-delay events execute
...     print time.time()
...
>>> print_some_times()
930343690.257
From print_time 930343695.274
From print_time 930343700.273
930343701.301
```
8.8.1. Scheduler Objects

`scheduler` instances have the following methods and attributes:

`scheduler.enterabs(time, priority, action, argument)`

Schedule a new event. The `time` argument should be a numeric type compatible with the return value of the `timefunc` function passed to the constructor. Events scheduled for the same `time` will be executed in the order of their `priority`.

Executing the event means executing `action(*argument)`. `argument` must be a sequence holding the parameters for `action`.

Return value is an event which may be used for later cancellation of the event (see `cancel()`).

`scheduler.enter(delay, priority, action, argument)`

Schedule an event for `delay` more time units. Other than the relative time, the other arguments, the effect and the return value are the same as those for `enterabs()`.

`scheduler.cancel(event)`

Remove the event from the queue. If `event` is not an event currently in the queue, this method will raise a `ValueError`.

`scheduler.empty()`

Return true if the event queue is empty.

`scheduler.run()`

Run all scheduled events. This function will wait (using the `delayfunc()` function passed to the constructor) for the next event, then execute it and so on until there are no more scheduled events.
Either action or delayfunc can raise an exception. In either case, the scheduler will maintain a consistent state and propagate the exception. If an exception is raised by action, the event will not be attempted in future calls to run().

If a sequence of events takes longer to run than the time available before the next event, the scheduler will simply fall behind. No events will be dropped; the calling code is responsible for canceling events which are no longer pertinent.

scheduler.queue
Read-only attribute returning a list of upcoming events in the order they will be run. Each event is shown as a named tuple with the following fields: time, priority, action, argument.

New in version 2.6.
8. Data Types
8.9. mutex — Mutual exclusion support

*Deprecated since version 2.6:* The `mutex` module has been removed in Python 3.

The `mutex` module defines a class that allows mutual-exclusion via acquiring and releasing locks. It does not require (or imply) `threading` or multi-tasking, though it could be useful for those purposes.

The `mutex` module defines the following class:

```python
class mutex
    Create a new (unlocked) mutex.
```

A mutex has two pieces of state — a “locked” bit and a queue. When the mutex is not locked, the queue is empty. Otherwise, the queue contains zero or more (function, argument) pairs representing functions (or methods) waiting to acquire the lock. When the mutex is unlocked while the queue is not empty, the first queue entry is removed and its function(argument) pair called, implying it now has the lock.

Of course, no multi-threading is implied – hence the funny interface for `lock()`, where a function is called once the lock is acquired.
8.9.1. Mutex Objects

**mutex** objects have following methods:

**mutex.test()**
Check whether the mutex is locked.

**mutex.testandset()**
“Atomic” test-and-set, grab the lock if it is not set, and return **True**, otherwise, return **False**.

**mutex.lock(function, argument)**
Execute **function(argument)**, unless the mutex is locked. In the case it is locked, place the function and argument on the queue. See **unlock()** for explanation of when **function(argument)** is executed in that case.

**mutex.unlock()**
Unlock the mutex if queue is empty, otherwise execute the first element in the queue.
8.10. queue — A synchronized queue class

**Note:** The `queue` module has been renamed to `queue` in Python 3. The `2to3` tool will automatically adapt imports when converting your sources to Python 3.

**Source code:** Lib/Queue.py

The `queue` module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The `Queue` class in this module implements all the required locking semantics. It depends on the availability of thread support in Python; see the `threading` module.

The module implements three types of queue, which differ only in the order in which the entries are retrieved. In a FIFO queue, the first tasks added are the first retrieved. In a LIFO queue, the most recently added entry is the first retrieved (operating like a stack). With a priority queue, the entries are kept sorted (using the `heapq` module) and the lowest valued entry is retrieved first.

The `queue` module defines the following classes and exceptions:

```python
class Queue: Queue(maxsize=0)
```

Constructor for a FIFO queue. `maxsize` is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If `maxsize` is less than or equal to zero, the queue size is infinite.
class Queue. **LifoQueue**(*maxsize=0*)

Constructor for a LIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

*New in version 2.6.*

class Queue. **PriorityQueue**(*maxsize=0*)

Constructor for a priority queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

The lowest valued entries are retrieved first (the lowest valued entry is the one returned by `sorted(list(entries))[0]`). A typical pattern for entries is a tuple in the form: (*priority_number*, *data*).

*New in version 2.6.*

exception Queue. **Empty**

Exception raised when non-blocking `get()` (or `get_nowait()`) is called on a Queue object which is empty.

exception Queue. **Full**

Exception raised when non-blocking `put()` (or `put_nowait()`) is called on a Queue object which is full.

**See also:** collections.deque is an alternative implementation of unbounded queues with fast atomic `append()` and `popleft()` operations that do not require locking.
8.10.1. Queue Objects

Queue objects (Queue, LifoQueue, or PriorityQueue) provide the public methods described below.

Queue.qsize()
Return the approximate size of the queue. Note, qsize() > 0 doesn't guarantee that a subsequent get() will not block, nor will qsize() < maxsize guarantee that put() will not block.

Queue.empty()
Return True if the queue is empty, False otherwise. If empty() returns True it doesn’t guarantee that a subsequent call to put() will not block. Similarly, if empty() returns False it doesn’t guarantee that a subsequent call to get() will not block.

Queue.full()
Return True if the queue is full, False otherwise. If full() returns True it doesn’t guarantee that a subsequent call to get() will not block. Similarly, if full() returns False it doesn’t guarantee that a subsequent call to put() will not block.

Queue.put(item[, block[, timeout]])
Put item into the queue. If optional args block is true and timeout is None (the default), block if necessary until a free slot is available. If timeout is a positive number, it blocks at most timeout seconds and raises the Full exception if no free slot was available within that time. Otherwise (block is false), put an item on the queue if a free slot is immediately available, else raise the Full exception (timeout is ignored in that case).

New in version 2.3: The timeout parameter.
Queue. put_nowait(item)

Equivalent to put(item, False).

Queue. get([block[, timeout]])

Remove and return an item from the queue. If optional args block is true and timeout is None (the default), block if necessary until an item is available. If timeout is a positive number, it blocks at most timeout seconds and raises the Empty exception if no item was available within that time. Otherwise (block is false), return an item if one is immediately available, else raise the Empty exception (timeout is ignored in that case).

New in version 2.3: The timeout parameter.

Queue. get_nowait()

Equivalent to get(False).

Two methods are offered to support tracking whether enqueued tasks have been fully processed by daemon consumer threads.

Queue. task_done()

Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each get() used to fetch a task, a subsequent call to task_done() tells the queue that the processing on the task is complete.

If a join() is currently blocking, it will resume when all items have been processed (meaning that a task_done() call was received for every item that had been put() into the queue).

Raises a ValueError if called more times than there were items placed in the queue.

New in version 2.5.
Queue.join()

Blocks until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls task_done() to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, join() unblocks.

New in version 2.5.

Example of how to wait for enqueued tasks to be completed:

```python
def worker():
    while True:
        item = q.get()
        do_work(item)
        q.task_done()

q = Queue()
for i in range(num_worker_threads):
    t = Thread(target=worker)
    t.daemon = True
    t.start()

for item in source():
    q.put(item)
q.join()  # block until all tasks are done
```
The `weakref` module allows the Python programmer to create *weak references* to objects.

In the following, the term *referent* means the object which is referred to by a weak reference.

A weak reference to an object is not enough to keep the object alive: when the only remaining references to a referent are weak references, *garbage collection* is free to destroy the referent and reuse its memory for something else. A primary use for weak references is to implement caches or mappings holding large objects, where it's desired that a large object not be kept alive solely because it appears in a cache or mapping.

For example, if you have a number of large binary image objects, you may wish to associate a name with each. If you used a Python dictionary to map names to images, or images to names, the image objects would remain alive just because they appeared as values or keys in the dictionaries. The `WeakKeyDictionary` and `WeakValueDictionary` classes supplied by the `weakref` module are an alternative, using weak references to construct mappings that don't keep objects alive solely because they appear in the mapping objects. If, for example, an image object is a value in a `WeakValueDictionary`, then when the last remaining references to that image object are the weak references held by weak mappings, garbage collection can reclaim the object, and its corresponding entries in weak mappings are simply deleted.
WeakKeyDictionary and WeakValueDictionary use weak references in their implementation, setting up callback functions on the weak references that notify the weak dictionaries when a key or value has been reclaimed by garbage collection. Most programs should find that using one of these weak dictionary types is all they need – it's not usually necessary to create your own weak references directly. The low-level machinery used by the weak dictionary implementations is exposed by the weakref module for the benefit of advanced uses.

Not all objects can be weakly referenced; those objects which can include class instances, functions written in Python (but not in C), methods (both bound and unbound), sets, frozensets, file objects, generators, type objects, DBcursor objects from the bsddb module, sockets, arrays, deques, regular expression pattern objects, and code objects.

*Changed in version 2.4:* Added support for files, sockets, arrays, and patterns.

*Changed in version 2.7:* Added support for thread.lock, threading.Lock, and code objects.

Several built-in types such as list and dict do not directly support weak references but can add support through subclassing:

```python
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)  # this object is weak reference
```

**CPython implementation detail:** Other built-in types such as tuple and long do not support weak references even when subclassed.
Extension types can easily be made to support weak references; see Weak Reference Support.

```python
class weakref.ref(object[, callback])
```

Return a weak reference to `object`. The original object can be retrieved by calling the reference object if the referent is still alive; if the referent is no longer alive, calling the reference object will cause `None` to be returned. If `callback` is provided and not `None`, and the returned weakref object is still alive, the callback will be called when the object is about to be finalized; the weak reference object will be passed as the only parameter to the callback; the referent will no longer be available.

It is allowable for many weak references to be constructed for the same object. Callbacks registered for each weak reference will be called from the most recently registered callback to the oldest registered callback.

Exceptions raised by the callback will be noted on the standard error output, but cannot be propagated; they are handled in exactly the same way as exceptions raised from an object’s `__del__()` method.

Weak references are `hashable` if the `object` is hashable. They will maintain their hash value even after the `object` was deleted. If `hash()` is called the first time only after the `object` was deleted, the call will raise `TypeError`.

Weak references support tests for equality, but not ordering. If the referents are still alive, two references have the same equality relationship as their referents (regardless of the `callback`). If either referent has been deleted, the references are equal only if the reference objects are the same object.
Changed in version 2.4: This is now a subclassable type rather than a factory function; it derives from \texttt{object}.

\begin{verbatim}
weaken\texttt{.proxy}(\texttt{object}[\texttt{, callback}\])
Return a proxy to \texttt{object} which uses a weak reference. This supports use of the proxy in most contexts instead of requiring the explicit dereferencing used with weak reference objects. The returned object will have a type of either \texttt{ProxyType} or \texttt{CallableProxyType}, depending on whether \texttt{object} is callable. Proxy objects are not \texttt{hashable} regardless of the referent; this avoids a number of problems related to their fundamentally mutable nature, and prevent their use as dictionary keys. \texttt{callback} is the same as the parameter of the same name to the \texttt{ref()} function.

\end{verbatim}

\begin{verbatim}
weaken\texttt{.getweakrefcount}(\texttt{object})
Return the number of weak references and proxies which refer to \texttt{object}.

\end{verbatim}

\begin{verbatim}
weaken\texttt{.getweakrefs}(\texttt{object})
Return a list of all weak reference and proxy objects which refer to \texttt{object}.

\end{verbatim}

\begin{verbatim}
class weaken\texttt{.WeakKeyDictionary}([\texttt{dict}])
Mapping class that references keys weakly. Entries in the dictionary will be discarded when there is no longer a strong reference to the key. This can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.

\end{verbatim}

\begin{verbatim}
\textbf{Note:} Caution: Because a \texttt{WeakKeyDictionary} is built on top of a Python dictionary, it must not change size when iterating over
\end{verbatim}
WeakKeyDictionary objects have the following additional methods. These expose the internal references directly. The references are not guaranteed to be “live” at the time they are used, so the result of calling the references needs to be checked before being used. This can be used to avoid creating references that will cause the garbage collector to keep the keys around longer than needed.

**WeakKeyDictionary**. `iterkeyrefs()`

Return an iterable of the weak references to the keys.

*New in version 2.5.*

**WeakKeyDictionary**. `keyrefs()`

Return a list of weak references to the keys.

*New in version 2.5.*

**class** `weakref.WeakValueDictionary([dict])`

Mapping class that references values weakly. Entries in the dictionary will be discarded when no strong reference to the value exists any more.

**Note:** Caution: Because a `WeakValueDictionary` is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a `WeakValueDictionary` because actions performed by the program during iteration may cause items in the dictionary to vanish “by magic” (as a side effect of garbage collection).
**WeakValueDictionary** objects have the following additional methods. These methods have the same issues as the `iterkeyrefs()` and `keyrefs()` methods of `WeakKeyDictionary` objects.

**WeakValueDictionary.**`itervaluerefs()`

Return an iterable of the weak references to the values.

*New in version 2.5.*

**WeakValueDictionary.**`valuerefs()`

Return a list of weak references to the values.

*New in version 2.5.*

class weakref.**WeakSet([elements])**

Set class that keeps weak references to its elements. An element will be discarded when no strong reference to it exists any more.

*New in version 2.7.*

**weakref.**`ReferenceType`

The type object for weak references objects.

**weakref.**`ProxyType`

The type object for proxies of objects which are not callable.

**weakref.**`CallableProxyType`

The type object for proxies of callable objects.

**weakref.**`ProxyTypes`

Sequence containing all the type objects for proxies. This can make it simpler to test if an object is a proxy without being dependent on naming both proxy types.

**exception** weakref.**ReferenceError**

Exception raised when a proxy object is used but the underlying
object has been collected. This is the same as the standard ReferenceError exception.

See also:

**PEP 205 - Weak References**

The proposal and rationale for this feature, including links to earlier implementations and information about similar features in other languages.
8.11.1. Weak Reference Objects

Weak reference objects have no attributes or methods, but do allow the referent to be obtained, if it still exists, by calling it:

```python
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

If the referent no longer exists, calling the reference object returns `None`:

```python
>>> del o, o2
>>> print r()
None
```

Testing that a weak reference object is still live should be done using the expression `ref() is not None`. Normally, application code that needs to use a reference object should follow this pattern:

```python
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print "Object has been deallocated; can't frobnicate."
else:
    print "Object is still live!"
    o.do_something_useful()
```

Using a separate test for “liveness” creates race conditions in threaded applications; another thread can cause a weak reference to become invalidated before the weak reference is called; the idiom
shown above is safe in threaded applications as well as single-threaded applications.

Specialized versions of \texttt{ref} objects can be created through subclassing. This is used in the implementation of the \texttt{WeakValueDictionary} to reduce the memory overhead for each entry in the mapping. This may be most useful to associate additional information with a reference, but could also be used to insert additional processing on calls to retrieve the referent.

This example shows how a subclass of \texttt{ref} can be used to store additional information about an object and affect the value that's returned when the referent is accessed:

```python
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, **annotations):
        super(ExtendedRef, self).__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def __call__(self):
        '''Return a pair containing the referent and the number times the reference has been called.
        '''
        ob = super(ExtendedRef, self).__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```
8.11.2. Example

This simple example shows how an application can use object IDs to retrieve objects that it has seen before. The IDs of the objects can then be used in other data structures without forcing the objects to remain alive, but the objects can still be retrieved by ID if they do.

```python
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```
The module defines a mixin, `DictMixin`, defining all dictionary methods for classes that already have a minimum mapping interface. This greatly simplifies writing classes that need to be substitutable for dictionaries (such as the shelve module).

This module also defines a class, `UserDict`, that acts as a wrapper around dictionary objects. The need for this class has been largely supplanted by the ability to subclass directly from `dict` (a feature that became available starting with Python version 2.2). Prior to the introduction of `dict`, the `UserDict` class was used to create dictionary-like sub-classes that obtained new behaviors by overriding existing methods or adding new ones.

The `UserDict` module defines the `UserDict` class and `DictMixin`:

```python
class UserDict

UserDict([initialdata])
```

Class that simulates a dictionary. The instance’s contents are kept in a regular dictionary, which is accessible via the `data` attribute of `UserDict` instances. If `initialdata` is provided, `data` is initialized with its contents; note that a reference to `initialdata` will not be kept, allowing it be used for other purposes.

**Note:** For backward compatibility, instances of `UserDict` are not iterable.

```python
class UserDict

IterableUserDict([initialdata])
```
Subclass of UserDict that supports direct iteration (e.g. for key in myDict).

In addition to supporting the methods and operations of mappings (see section Mapping Types — dict), UserDict and IterableUserDict instances provide the following attribute:

IterableUserDict.data

A real dictionary used to store the contents of the UserDict class.

class UserDict.DictMixin

Mixin defining all dictionary methods for classes that already have a minimum dictionary interface including __getitem__(), __setitem__(), __delitem__(), and keys().

This mixin should be used as a superclass. Adding each of the above methods adds progressively more functionality. For instance, defining all but __delitem__() will preclude only pop() and popitem() from the full interface.

In addition to the four base methods, progressively more efficiency comes with defining __contains__(), __iter__(), and iteritems().

Since the mixin has no knowledge of the subclass constructor, it does not define __init__() or copy().

Starting with Python version 2.6, it is recommended to use collections.MutableMapping instead of DictMixin.

Note that DictMixin does not implement the viewkeys(), viewvalues(), or viewitems() methods.
8.13. **UserList** — Class wrapper for list objects

**Note:** When Python 2.2 was released, many of the use cases for this class were subsumed by the ability to subclass `list` directly. However, a handful of use cases remain.

This module provides a list-interface around an underlying data store. By default, that data store is a `list`; however, it can be used to wrap a list-like interface around other objects (such as persistent storage).

In addition, this class can be mixed-in with built-in classes using multiple inheritance. This can sometimes be useful. For example, you can inherit from `UserList` and `str` at the same time. That would not be possible with both a real `list` and a real `str`.

This module defines a class that acts as a wrapper around list objects. It is a useful base class for your own list-like classes, which can inherit from them and override existing methods or add new ones. In this way one can add new behaviors to lists.

The `UserList` module defines the `UserList` class:

```python
class UserList(UserList):
    def __init__(self, list):
        super().__init__(list)
```

Class that simulates a list. The instance's contents are kept in a regular list, which is accessible via the `data` attribute of `UserList` instances. The instance's contents are initially set to a copy of `list`, defaulting to the empty list `[]`. `list` can be any iterable, e.g. a real Python list or a `UserList` object.

**Note:** The `UserList` class has been moved to the `collections` module.
module in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

In addition to supporting the methods and operations of mutable sequences (see section Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange), UserList instances provide the following attribute:

UserList.data

A real Python list object used to store the contents of the UserList class.

**Subclassing requirements:** Subclasses of UserList are expected to offer a constructor which can be called with either no arguments or one argument. List operations which return a new sequence attempt to create an instance of the actual implementation class. To do so, it assumes that the constructor can be called with a single parameter, which is a sequence object used as a data source.

If a derived class does not wish to comply with this requirement, all of the special methods supported by this class will need to be overridden; please consult the sources for information about the methods which need to be provided in that case.

*Changed in version 2.0:* Python versions 1.5.2 and 1.6 also required that the constructor be callable with no parameters, and offer a mutable data attribute. Earlier versions of Python did not attempt to create instances of the derived class.
8.14. **UserString** — Class wrapper for string objects

**Note:** This `UserString` class from this module is available for backward compatibility only. If you are writing code that does not need to work with versions of Python earlier than Python 2.2, please consider subclassing directly from the built-in `str` type instead of using `UserString` (there is no built-in equivalent to `MutableString`).

This module defines a class that acts as a wrapper around string objects. It is a useful base class for your own string-like classes, which can inherit from them and override existing methods or add new ones. In this way one can add new behaviors to strings.

It should be noted that these classes are highly inefficient compared to real string or Unicode objects; this is especially the case for `MutableString`.

The `UserString` module defines the following classes:

```python
class UserString.
```

`UserString(sequence)`

Class that simulates a string or a Unicode string object. The instance's content is kept in a regular string or Unicode string object, which is accessible via the `data` attribute of `UserString` instances. The instance's contents are initially set to a copy of `sequence`. `sequence` can be either a regular Python string or Unicode string, an instance of `UserString` (or a subclass) or an arbitrary sequence which can be converted into a string using the built-in `str()` function.
Note: The UserString class has been moved to the collections module in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

class UserString.**MutableString**(sequence)

This class is derived from the UserString above and redefines strings to be mutable. Mutable strings can't be used as dictionary keys, because dictionaries require immutable objects as keys. The main intention of this class is to serve as an educational example for inheritance and necessity to remove (override) the __hash__() method in order to trap attempts to use a mutable object as dictionary key, which would be otherwise very error prone and hard to track down.

*Deprecated since version 2.6:* The **MutableString** class has been removed in Python 3.

In addition to supporting the methods and operations of string and Unicode objects (see section String Methods), UserString instances provide the following attribute:

**MutableString.**data

A real Python string or Unicode object used to store the content of the UserString class.
This module defines names for some object types that are used by the standard Python interpreter, but not for the types defined by various extension modules. Also, it does not include some of the types that arise during processing such as the `listiterator` type. It is safe to use `from types import *` — the module does not export any names besides the ones listed here. New names exported by future versions of this module will all end in `Type`.

Typical use is for functions that do different things depending on their argument types, like the following:

```python
from types import *
def delete(mylist, item):
    if type(item) is IntType:
        del mylist[item]
    else:
        mylist.remove(item)
```

Starting in Python 2.2, built-in factory functions such as `int()` and `str()` are also names for the corresponding types. This is now the preferred way to access the type instead of using the `types` module. Accordingly, the example above should be written as follows:

```python
def delete(mylist, item):
    if isinstance(item, int):
        del mylist[item]
    else:
        mylist.remove(item)
```
The module defines the following names:

types. **NoneType**  
    The type of `None`.

types. **TypeType**  
    The type of type objects (such as returned by `type()`); alias of the built-in `type`.

types. **BooleanType**  
    The type of the `bool` values `True` and `False`; alias of the built-in `bool`.

    *New in version 2.3.*

types. **IntType**  
    The type of integers (e.g. `1`); alias of the built-in `int`.

types. **LongType**  
    The type of long integers (e.g. `1L`); alias of the built-in `long`.

types. **FloatType**  
    The type of floating point numbers (e.g. `1.0`); alias of the built-in `float`.

types. **ComplexType**  
    The type of complex numbers (e.g. `1.0j`). This is not defined if Python was built without complex number support.

types. **StringType**  
    The type of character strings (e.g. `'Spam'`); alias of the built-in `str`.

types. **UnicodeType**  
    The type of Unicode character strings (e.g. `u'Spam'`). This is not defined if Python was built without Unicode support. It's an alias
of the built-in `unicode`.

types. **TupleType**  
The type of tuples (e.g. `(1, 2, 3, 'Spam')`); alias of the built-in `tuple`.

types. **ListType**  
The type of lists (e.g. `[0, 1, 2, 3]`); alias of the built-in `list`.

types. **DictType**  
The type of dictionaries (e.g. `{'Bacon': 1, 'Ham': 0}`); alias of the built-in `dict`.

types. **DictionaryType**  
An alternate name for `DictType`.

types. **FunctionType**  
types. **LambdaType**  
The type of user-defined functions and functions created by `lambda` expressions.

types. **GeneratorType**  
The type of generator-iterator objects, produced by calling a generator function.  

*New in version 2.2.*

types. **CodeType**  
The type for code objects such as returned by `compile()`.

types. **ClassType**  
The type of user-defined old-style classes.

types. **InstanceType**  
The type of instances of user-defined old-style classes.

types. **MethodType**
The type of methods of user-defined class instances.

types. **UnboundMethodType**

An alternate name for **MethodType**.

types. **BuiltinFunctionType**
types. **BuiltinMethodType**

The type of built-in functions like `len()` or `sys.exit()`, and methods of built-in classes. (Here, the term “built-in” means “written in C”.)

types. **ModuleType**

The type of modules.

types. **FileType**

The type of open file objects such as `sys.stdout`; alias of the built-in `file`.

types. **XRangeType**

The type of range objects returned by `xrange()`; alias of the built-in `xrange`.

types. **SliceType**

The type of objects returned by `slice()`; alias of the built-in `slice`.

types. **EllipsisType**

The type of **Ellipsis**.

types. **TracebackType**

The type of traceback objects such as found in `sys.exc_traceback`.

types. **FrameType**

The type of frame objects such as found in `tb.tb_frame` if `tb` is a traceback object.
BufferType
The type of buffer objects created by the buffer() function.

DictProxyType
The type of dict proxies, such as TypeType.__dict__.

NotImplementedType
The type of NotImplemented

GetSetDescriptorType
The type of objects defined in extension modules with PyGetSetDef, such as FrameType.f_locals or array.array.typecode. This type is used as descriptor for object attributes; it has the same purpose as the property type, but for classes defined in extension modules.

New in version 2.5.

MemberDescriptorType
The type of objects defined in extension modules with PyMemberDef, such as datetime.timedelta.days. This type is used as descriptor for simple C data members which use standard conversion functions; it has the same purpose as the property type, but for classes defined in extension modules.

CPython implementation detail: In other implementations of Python, this type may be identical to GetSetDescriptorType.

New in version 2.5.

StringTypes
A sequence containing StringType and UnicodeType used to facilitate easier checking for any string object. Using this is more portable than using a sequence of the two string types
constructed elsewhere since it only contains UnicodeType if it has been built in the running version of Python. For example: `isinstance(s, types.StringTypes)`. 

*New in version 2.2.*
8.16. new — Creation of runtime internal objects

*Deprecated since version 2.6:* The `new` module has been removed in Python 3. Use the `types` module’s classes instead.

The `new` module allows an interface to the interpreter object creation functions. This is for use primarily in marshal-type functions, when a new object needs to be created “magically” and not by using the regular creation functions. This module provides a low-level interface to the interpreter, so care must be exercised when using this module. It is possible to supply non-sensical arguments which crash the interpreter when the object is used.

The `new` module defines the following functions:

new. `instance(class[, dict])`

This function creates an instance of `class` with dictionary `dict` without calling the `__init__()` constructor. If `dict` is omitted or `None`, a new, empty dictionary is created for the new instance. Note that there are no guarantees that the object will be in a consistent state.

new. `instancetype(function, instance, class)`

This function will return a method object, bound to `instance`, or unbound if `instance` is `None`. `function` must be callable.

new. `function(code, globals[, name[, argdefs[, closure]]]])`

Returns a (Python) function with the given code and globals. If `name` is given, it must be a string or `None`. If it is a string, the function will have the given name, otherwise the function name
will be taken from `code.co_name`. If `argdefs` is given, it must be a tuple and will be used to determine the default values of parameters. If `closure` is given, it must be `None` or a tuple of cell objects containing objects to bind to the names in `code.co_freevars`.

```python
new. code(argcount, nlocals, stacksize, flags, codestring, constants, names, varnames, filename, name, firstlineno, lnotab)
```

This function is an interface to the `PyCode_New()` C function.

```python
new. module(name[, doc])
```

This function returns a new module object with name `name`. `name` must be a string. The optional `doc` argument can have any type.

```python
new. classobj(name, baseclasses, dict)
```

This function returns a new class object, with name `name`, derived from `baseclasses` (which should be a tuple of classes) and with namespace `dict`. 
8.17. copy — Shallow and deep copy operations

Assignment statements in Python do not copy objects, they create bindings between a target and an object. For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy without changing the other. This module provides generic shallow and deep copy operations (explained below).

Interface summary:

```python
copy.copy(x)

Return a shallow copy of x.

copy.deepcopy(x)

Return a deep copy of x.

exception copy.error

Raised for module specific errors.
```

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
- A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

Two problems often exist with deep copy operations that don’t exist with shallow copy operations:
Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop.

Because deep copy copies *everything* it may copy too much, e.g., even administrative data structures that should be shared even between copies.

The `deepcopy()` function avoids these problems by:

- keeping a “memo” dictionary of objects already copied during the current copying pass; and
- letting user-defined classes override the copying operation or the set of components copied.

This module does not copy types like module, method, stack trace, stack frame, file, socket, window, array, or any similar types. It does “copy” functions and classes (shallow and deeply), by returning the original object unchanged; this is compatible with the way these are treated by the `pickle` module.

Shallow copies of dictionaries can be made using `dict.copy()`, and of lists by assigning a slice of the entire list, for example, `copied_list = original_list[:]`.

*Changed in version 2.5:* Added copying functions.

Classes can use the same interfaces to control copying that they use to control pickling. See the description of module `pickle` for information on these methods. The `copy` module does not use the `copy_reg` registration module.

In order for a class to define its own copy implementation, it can define special methods `__copy__()` and `__deepcopy__()`. The former is called to implement the shallow copy operation; no additional arguments are passed. The latter is called to implement the deep copy operation; it is passed one argument, the memo dictionary. If
the `__deepcopy__()` implementation needs to make a deep copy of a component, it should call the `deepcopy()` function with the component as first argument and the memo dictionary as second argument.

**See also:**

**Module pickle**

Discussion of the special methods used to support object state retrieval and restoration.
8.18. pprint — Data pretty printer

**Source code:** Lib/pprint.py

The pprint module provides a capability to “pretty-print” arbitrary Python data structures in a form which can be used as input to the interpreter. If the formatted structures include objects which are not fundamental Python types, the representation may not be loadable. This may be the case if objects such as files, sockets, classes, or instances are included, as well as many other built-in objects which are not representable as Python constants.

The formatted representation keeps objects on a single line if it can, and breaks them onto multiple lines if they don't fit within the allowed width. Construct **PrettyPrinter** objects explicitly if you need to adjust the width constraint.

*Changed in version 2.5:* Dictionaries are sorted by key before the display is computed; before 2.5, a dictionary was sorted only if its display required more than one line, although that wasn’t documented.

*Changed in version 2.6:* Added support for **set** and **frozenset**.

The pprint module defines one class:

```python
class pprint.PrettyPrinter(indent=1, width=80, depth=None, stream=None)
```

Construct a **PrettyPrinter** instance. This constructor understands several keyword parameters. An output stream may be set using the **stream** keyword; the only method used on the stream object is the file protocol’s **write()** method. If not specified, the **PrettyPrinter** adopts **sys.stdout**. Three additional
parameters may be used to control the formatted representation. The keywords are `indent`, `depth`, and `width`. The amount of indentation added for each recursive level is specified by `indent`; the default is one. Other values can cause output to look a little odd, but can make nesting easier to spot. The number of levels which may be printed is controlled by `depth`; if the data structure being printed is too deep, the next contained level is replaced by `...`. By default, there is no constraint on the depth of the objects being formatted. The desired output width is constrained using the `width` parameter; the default is 80 characters. If a structure cannot be formatted within the constrained width, a best effort will be made.

```python
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
['spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni'
 ...
 ('parrot', ('fresh fruit',))))))))
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
...

The `PrettyPrinter` class supports several derivative functions:

```
pprint.pformat(object, indent=1, width=80, depth=None)
```

Return the formatted representation of `object` as a string. `indent`, `width` and `depth` will be passed to the `PrettyPrinter` constructor as formatting parameters.

*Changed in version 2.4:* The parameters `indent`, `width` and `depth`
were added.

\texttt{pprint\.(object, stream=None, indent=1, width=80, depth=None)}

Prints the formatted representation of \textit{object} on \textit{stream}, followed by a newline. If \textit{stream} is \texttt{None}, \texttt{sys.stdout} is used. This may be used in the interactive interpreter instead of a \texttt{print} statement for inspecting values. \textit{indent}, \textit{width} and \textit{depth} will be passed to the \texttt{PrettyPrinter} constructor as formatting parameters.

\begin{verbatim}
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
\end{verbatim}

\textit{Changed in version 2.4}: The parameters \textit{indent}, \textit{width} and \textit{depth} were added.

\texttt{pprint\.(isreadable\(\)(object))}

Determine if the formatted representation of \textit{object} is “readable,” or can be used to reconstruct the value using \texttt{eval()}. This always returns \texttt{False} for recursive objects.

\begin{verbatim}
>>> pprint.isreadable(stuff)
False
\end{verbatim}

\texttt{pprint\.(isrecursive\(\)(object))}

Determine if \textit{object} requires a recursive representation.

One more support function is also defined:

\texttt{pprint\.(saferepr\(\)(object))}
Return a string representation of object, protected against recursive data structures. If the representation of object exposes a recursive entry, the recursive reference will be represented as <Recursion on typename with id=number>. The representation is not otherwise formatted.

```python
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'ni']"
```
8.18.1. PrettyPrinter Objects

PrettyPrinter instances have the following methods:

PrettyPrinter. **pformat**(object)
Return the formatted representation of object. This takes into account the options passed to the PrettyPrinter constructor.

PrettyPrinter. **pprint**(object)
Print the formatted representation of object on the configured stream, followed by a newline.

The following methods provide the implementations for the corresponding functions of the same names. Using these methods on an instance is slightly more efficient since new PrettyPrinter objects don’t need to be created.

PrettyPrinter. **isreadable**(object)
Determine if the formatted representation of the object is “readable,” or can be used to reconstruct the value using `eval()`. Note that this returns `False` for recursive objects. If the depth parameter of the PrettyPrinter is set and the object is deeper than allowed, this returns `False`.

PrettyPrinter. **isrecursive**(object)
Determine if the object requires a recursive representation.

This method is provided as a hook to allow subclasses to modify the way objects are converted to strings. The default implementation uses the internals of the saferepr() implementation.

PrettyPrinter. **format**(object, context, maxlevels, level)
Returns three values: the formatted version of object as a string,
a flag indicating whether the result is readable, and a flag indicating whether recursion was detected. The first argument is the object to be presented. The second is a dictionary which contains the \texttt{id()} of objects that are part of the current presentation context (direct and indirect containers for object that are affecting the presentation) as the keys; if an object needs to be presented which is already represented in context, the third return value should be \texttt{True}. Recursive calls to the \texttt{format()} method should add additional entries for containers to this dictionary. The third argument, \texttt{maxlevels}, gives the requested limit to recursion; this will be \texttt{0} if there is no requested limit. This argument should be passed unmodified to recursive calls. The fourth argument, \texttt{level}, gives the current level; recursive calls should be passed a value less than that of the current call.

\textit{New in version 2.3.}
8.18.2. pprint Example

This example demonstrates several uses of the `pprint()` function and its parameters.

```python
>>> import pprint
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', (...

... ('parrot', ('fresh fruit',))))))))
>>> stuff = ['a' * 10, tup, ['a' * 30, 'b' * 30, 'c' * 20, 'd'

>>> pprint.pprint(stuff)
['aaaaaaaaaa',
 ('spam',
  ('eggs',
   ('lumberjack',
    ('knights', ('ni', ('dead', ('parrot', ('fresh fruit',)))))))))
['aaaaaaaaaaaaaaaaaaaaaaaaaaaa', 'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb'
 ['ccccccccccccccccccccccc', 'ddddddeeeeffggggggggg']

>>> pprint.pprint(stuff, depth=3)
['aaaaaaaaa',
 ('spam', ('eggs', (...))),
 ['aaaaaaaaaaaaaaaaaaaaaaaaaaaa', 'bbbbbbbbbbbbbbbbbbbbbbbbbb'
 ['cccccccccccccccccccccc', 'ddddddeeeeffggggggggg']

>>> pprint.pprint(stuff, width=60)
['aaaaaaaaaa',
 ('spam',
  ('eggs',
   ('lumberjack',
    ('knights',
     ('ni', ('dead', ('parrot', ('fresh fruit',))))))))),
 ['aaaaaaaaaaaaaaaaaaaaaaaaaaaaa', 'bbbbbbbbbbbbbbbbbbbbbbbbb',
 ['cccccccccccccccccccccc', 'ddddddeeeeffggggggggg']]
```
### 8.19. `repr` — Alternate `repr()` implementation

**Note:** The `repr` module has been renamed to `reprlib` in Python 3. The `2to3` tool will automatically adapt imports when converting your sources to Python 3.

**Source code:** `Lib/repr.py`

The `repr` module provides a means for producing object representations with limits on the size of the resulting strings. This is used in the Python debugger and may be useful in other contexts as well.

This module provides a class, an instance, and a function:

```python
class repr.Repr
    Class which provides formatting services useful in implementing functions similar to the built-in `repr()`; size limits for different object types are added to avoid the generation of representations which are excessively long.

repr.aRepr
    This is an instance of `Repr` which is used to provide the `repr()` function described below. Changing the attributes of this object will affect the size limits used by `repr()` and the Python debugger.

repr.repr(obj)
    This is the `repr()` method of `aRepr`. It returns a string similar to that returned by the built-in function of the same name, but with limits on most sizes.
```
8.19.1. Repr Objects

Repr instances provide several attributes which can be used to provide size limits for the representations of different object types, and methods which format specific object types.

Repr. **maxlevel**

Depth limit on the creation of recursive representations. The default is 6.

Repr. **maxdict**
Repr. **maxlist**
Repr. **maxtuple**
Repr. **maxset**
Repr. **maxfrozenset**
Repr. **maxdeque**
Repr. **maxarray**

Limits on the number of entries represented for the named object type. The default is 4 for **maxdict**, 5 for **maxarray**, and 6 for the others.

*New in version 2.4:* **maxset**, **maxfrozenset**, and **set**.

Repr. **maxlong**

Maximum number of characters in the representation for a long integer. Digits are dropped from the middle. The default is 40.

Repr. **maxstring**

Limit on the number of characters in the representation of the string. Note that the “normal” representation of the string is used as the character source: if escape sequences are needed in the representation, these may be mangled when the representation is shortened. The default is 30.

Repr. **maxother**
This limit is used to control the size of object types for which no specific formatting method is available on the `Repr` object. It is applied in a similar manner as `maxstring`. The default is 20.

Repr. `repr(obj)`

The equivalent to the built-in `repr()` that uses the formatting imposed by the instance.

Repr. `repr1(obj, level)`

Recursive implementation used by `repr()`. This uses the type of `obj` to determine which formatting method to call, passing it `obj` and `level`. The type-specific methods should call `repr1()` to perform recursive formatting, with `level - 1` for the value of `level` in the recursive call.

Repr. `repr_TYPE(obj, level)`

Formatting methods for specific types are implemented as methods with a name based on the type name. In the method name, `TYPE` is replaced by `string.join(string.split(type(obj).__name__, '_'))`. Dispatch to these methods is handled by `repr1()`. Type-specific methods which need to recursively format a value should call `self.repr1(subobj, level - 1)`. 
8.19.2. Subclassing Repr Objects

The use of dynamic dispatching by `Repr.repr1()` allows subclasses of `Repr` to add support for additional built-in object types or to modify the handling of types already supported. This example shows how special support for file objects could be added:

```python
import repr as reprlib
import sys

class MyRepr(reprlib.Repr):
    def repr_file(self, obj, level):
        if obj.name in ['<stdin>', '<stdout>', '<stderr>']:
            return obj.name
        else:
            return repr(obj)

aRepr = MyRepr()
print aRepr.repr(sys.stdin)           # prints '<stdin>'
```

© Copyright 1990-2016, Python Software Foundation.
The Python Software Foundation is a non-profit corporation.
Please donate.
Last updated on Dec 17, 2016.
Found a bug?
Created using Sphinx 1.4.4.
9. Numeric and Mathematical Modules

The modules described in this chapter provide numeric and math-related functions and data types. The numbers module defines an abstract hierarchy of numeric types. The math and cmath modules contain various mathematical functions for floating-point and complex numbers. For users more interested in decimal accuracy than in speed, the decimal module supports exact representations of decimal numbers.

The following modules are documented in this chapter:

- **9.1. numbers — Numeric abstract base classes**
  - 9.1.1. The numeric tower
  - 9.1.2. Notes for type implementors
    - 9.1.2.1. Adding More Numeric ABCs
    - 9.1.2.2. Implementing the arithmetic operations
- **9.2. math — Mathematical functions**
  - 9.2.1. Number-theoretic and representation functions
  - 9.2.2. Power and logarithmic functions
  - 9.2.3. Trigonometric functions
  - 9.2.4. Angular conversion
  - 9.2.5. Hyperbolic functions
  - 9.2.6. Special functions
  - 9.2.7. Constants
- **9.3. cmath — Mathematical functions for complex numbers**
  - 9.3.1. Conversions to and from polar coordinates
  - 9.3.2. Power and logarithmic functions
  - 9.3.3. Trigonometric functions
  - 9.3.4. Hyperbolic functions
  - 9.3.5. Classification functions
- 9.3.6. Constants
- 9.4. **decimal** — Decimal fixed point and floating point arithmetic
  - 9.4.1. Quick-start Tutorial
  - 9.4.2. Decimal objects
    - 9.4.2.1. Logical operands
  - 9.4.3. Context objects
  - 9.4.4. Signals
  - 9.4.5. Floating Point Notes
    - 9.4.5.1. Mitigating round-off error with increased precision
    - 9.4.5.2. Special values
  - 9.4.6. Working with threads
  - 9.4.7. Recipes
  - 9.4.8. Decimal FAQ
- 9.5. **fractions** — Rational numbers
- 9.6. **random** — Generate pseudo-random numbers
- 9.7. **itertools** — Functions creating iterators for efficient looping
  - 9.7.1. Itertool functions
  - 9.7.2. Recipes
- 9.8. **functools** — Higher-order functions and operations on callable objects
  - 9.8.1. **partial** Objects
- 9.9. **operator** — Standard operators as functions
  - 9.9.1. Mapping Operators to Functions
9.1. numbers — Numeric abstract base classes

New in version 2.6.

The numbers module (PEP 3141) defines a hierarchy of numeric abstract base classes which progressively define more operations. None of the types defined in this module can be instantiated.

class numbers.Number

  The root of the numeric hierarchy. If you just want to check if an argument \( x \) is a number, without caring what kind, use
  
  `isinstance(x, Number)`. 
9.1.1. The numeric tower

class numbers. Complex

Subclasses of this type describe complex numbers and include
the operations that work on the built-in complex type. These are:
conversions to complex and bool, real, imag, +, -, *, /, abs(),
conjugate(), ==, and !=. All except - and != are abstract.

real

Abstract. Retrieves the real component of this number.

imag

Abstract. Retrieves the imaginary component of this number.

conjugate()

Abstract. Returns the complex conjugate. For example,
(1+3j).conjugate() == (1-3j).

class numbers. Real

To Complex, Real adds the operations that work on real numbers.

In short, those are: a conversion to float, math.trunc(), round(),
math.floor(), math.ceil(), divmod(), //, %, <, <=, >, and >=.

Real also provides defaults for complex(), real, imag, and
conjugate().

class numbers. Rational

Subtypes Real and adds numerator and denominator properties,
which should be in lowest terms. With these, it provides a default
for float().

numerator
Abstract.

**denominator**

Abstract.

```text
class numbers. **Integral**
```

Subtypes **Rational** and adds a conversion to **int**. Provides defaults for **float()**, **numerator**, and **denominator**. Adds abstract methods for **** and bit-string operations: <<, >>, &, ^, |, ~.
Implementors should be careful to make equal numbers equal and hash them to the same values. This may be subtle if there are two different extensions of the real numbers. For example, \texttt{fractions.Fraction} implements \texttt{hash()} as follows:

```python
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on
        # simple fractions.
        return hash((self.numerator, self.denominator))
```

9.1.2.1. Adding More Numeric ABCs

There are, of course, more possible ABCs for numbers, and this would be a poor hierarchy if it precluded the possibility of adding those. You can add \texttt{MyFoo} between \texttt{Complex} and \texttt{Real} with:

```python
class MyFoo(Complex): ...
MyFoo.register(Real)
```

9.1.2.2. Implementing the arithmetic operations

We want to implement the arithmetic operations so that mixed-mode operations either call an implementation whose author knew about the types of both arguments, or convert both to the nearest built in type and do the operation there. For subtypes of \texttt{Integral}, this means that \texttt{__add__()} and \texttt{__radd__()} should be defined as:
There are 5 different cases for a mixed-type operation on subclasses of `Complex`. I'll refer to all of the above code that doesn't refer to `MyIntegral` and `OtherTypeIKnowAbout` as "boilerplate". `a` will be an instance of `A`, which is a subtype of `Complex` (`a : A <: Complex`), and `b : B <: Complex`. I'll consider `a + b`:

1. If `A` defines an `__add__()` which accepts `b`, all is well.
2. If `A` falls back to the boilerplate code, and it were to return a value from `__add__()`, we’d miss the possibility that `B` defines a more intelligent `__radd__()`, so the boilerplate should return `NotImplemented` from `__add__()`. (Or `A` may not implement `__add__()` at all.)
3. Then `B`'s `__radd__()` gets a chance. If it accepts `a`, all is well.
4. If it falls back to the boilerplate, there are no more possible
methods to try, so this is where the default implementation should live.

5. If $B <: A$, Python tries $B.__radd__$ before $A.__add__$. This is ok, because it was implemented with knowledge of $A$, so it can handle those instances before delegating to $\texttt{Complex}$.

If $A <: \texttt{Complex}$ and $B <: \texttt{Real}$ without sharing any other knowledge, then the appropriate shared operation is the one involving the built in $\texttt{complex}$, and both $\texttt{__radd__()}$s land there, so $a+b == b+a$.

Because most of the operations on any given type will be very similar, it can be useful to define a helper function which generates the forward and reverse instances of any given operator. For example, $\texttt{fractions.Fraction}$ uses:

```python
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, long, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
        forward.__name__ = '__' + fallback_operator.__name__ + '__'
        forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
            return monomorphic_operator(a, b)
        elif isinstance(a, numbers.Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, numbers.Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
        reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
        reverse.__doc__ = monomorphic_operator.__doc__
```

```
return forward, reverse

def _add(a, b):
    """a + b""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)
```

9. Numeric and Mathematical Modules

The Python Standard Library
This module is always available. It provides access to the mathematical functions defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the \texttt{cmath} module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don’t is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.
9.2.1. Number-theoretic and representation functions

`math.ceil(x)`
Return the ceiling of \( x \) as a float, the smallest integer value greater than or equal to \( x \).

`math.copysign(x, y)`
Return \( x \) with the sign of \( y \). On a platform that supports signed zeros, `copysign(1.0, -0.0)` returns \(-1.0\).

*New in version 2.6.*

`math.fabs(x)`
Return the absolute value of \( x \).

`math.factorial(x)`
Return \( x \) factorial. Raises `ValueError` if \( x \) is not integral or is negative.

*New in version 2.6.*

`math.floor(x)`
Return the floor of \( x \) as a float, the largest integer value less than or equal to \( x \).

`math.fmod(x, y)`
Return \( fmod(x, y) \), as defined by the platform C library. Note that the Python expression \( x \% y \) may not return the same result. The intent of the C standard is that \( fmod(x, y) \) be exactly (mathematically; to infinite precision) equal to \( x - n* y \) for some integer \( n \) such that the result has the same sign as \( x \) and magnitude less than \( \text{abs}(y) \). Python’s \( x \% y \) returns a result with
the sign of \( y \) instead, and may not be exactly computable for float arguments. For example, \( \text{fmod}(-1e-100, 1e100) \) is \(-1e-100\), but the result of Python’s \(-1e-100 \% 1e100\) is \(1e100-1e-100\), which cannot be represented exactly as a float, and rounds to the surprising \(1e100\). For this reason, function \( \text{fmod}() \) is generally preferred when working with floats, while Python’s \( x \% y \) is preferred when working with integers.

**math. \texttt{frexp}(x)\texttt{**}

Return the mantissa and exponent of \( x \) as the pair \((m, e)\). \( m \) is a float and \( e \) is an integer such that \( x = m \times 2^{e} \) exactly. If \( x \) is zero, returns \((0.0, 0)\), otherwise \(0.5 \leq \text{abs}(m) < 1\). This is used to “pick apart” the internal representation of a float in a portable way.

**math. \texttt{fsum}(iterable)\texttt{**}

Return an accurate floating point sum of values in the iterable. Avoids loss of precision by tracking multiple intermediate partial sums:

```python
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1])
1.0
```

The algorithm’s accuracy depends on IEEE-754 arithmetic guarantees and the typical case where the rounding mode is half-even. On some non-Windows builds, the underlying C library uses extended precision addition and may occasionally double-round an intermediate sum causing it to be off in its least significant bit.

For further discussion and two alternative approaches, see the [ASPN cookbook recipes for accurate floating point summation](https://docs.python.org/3/library/math.html#math.frexp).
New in version 2.6.

```
math.isinf(x)
```

Check if the float \( x \) is positive or negative infinity.

New in version 2.6.

```
math.isnan(x)
```

Check if the float \( x \) is a NaN (not a number). For more information on NaNs, see the IEEE 754 standards.

New in version 2.6.

```
math.ldexp(x, i)
```

Return \( x \times (2**i) \). This is essentially the inverse of function \( \text{frexp()} \).

```
math.modf(x)
```

Return the fractional and integer parts of \( x \). Both results carry the sign of \( x \) and are floats.

```
math.trunc(x)
```

Return the **Real** value \( x \) truncated to an **Integral** (usually a long integer). Uses the **_trunc_** method.

New in version 2.6.

Note that \( \text{frexp()} \) and \( \text{modf()} \) have a different call/return pattern than their C equivalents: they take a single argument and return a pair of values, rather than returning their second return value through an ‘output parameter’ (there is no such thing in Python).

For the \( \text{ceil()}, \text{floor()} \), and \( \text{modf()} \) functions, note that all floating-point numbers of sufficiently large magnitude are exact integers. Python floats typically carry no more than 53 bits of precision (the same as the platform C double type), in which case any float \( x \) with
abs(x) \geq 2^{**52} necessarily has no fractional bits.
9.2.2. Power and logarithmic functions

math. \texttt{exp}(x)
Return \(e^x\).

math. \texttt{expm1}(x)
Return \(e^x - 1\). For small floats \(x\), the subtraction in \(\texttt{exp}(x) - 1\) can result in a significant loss of precision; the \texttt{expm1()} function provides a way to compute this quantity to full precision:

```python
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

*New in version 2.7.*

math. \texttt{log}(x[, base])
With one argument, return the natural logarithm of \(x\) (to base \(e\)).

With two arguments, return the logarithm of \(x\) to the given \texttt{base}, calculated as \(\log(x)/\log(\texttt{base})\).

*Changed in version 2.3: base argument added.*

math. \texttt{log1p}(x)
Return the natural logarithm of \(1+x\) (base \(e\)). The result is calculated in a way which is accurate for \(x\) near zero.

*New in version 2.6.*

math. \texttt{log10}(x)
Return the base-10 logarithm of \(x\). This is usually more accurate than \(\log(x, 10)\).
math. **pow**(x, y)

Return x raised to the power y. Exceptional cases follow Annex ‘F’ of the C99 standard as far as possible. In particular, **pow**(1.0, x) and **pow**(x, 0.0) always return 1.0, even when x is a zero or a NaN. If both x and y are finite, x is negative, and y is not an integer then **pow**(x, y) is undefined, and raises **ValueError**.

Unlike the built-in ** operator, **math.pow()** converts both its arguments to type **float**. Use ** or the built-in **pow()** function for computing exact integer powers.

**Changed in version 2.6:** The outcome of 1**nan** and **nan****0** was undefined.

math. **sqrt**(x)

Return the square root of x.
9.2.3. Trigonometric functions

- `math.acos(x)`
  Return the arc cosine of \( x \), in radians.

- `math.asin(x)`
  Return the arc sine of \( x \), in radians.

- `math.atan(x)`
  Return the arc tangent of \( x \), in radians.

- `math.atan2(y, x)`
  Return \( \text{atan}(y / x) \), in radians. The result is between \(-\pi\) and \(\pi\). The vector in the plane from the origin to point \((x, y)\) makes this angle with the positive X axis. The point of `atan2()` is that the signs of both inputs are known to it, so it can compute the correct quadrant for the angle. For example, `atan(1)` and `atan2(1, 1)` are both \(\pi/4\), but `atan2(-1, -1)` is \(-3\pi/4\).

- `math.cos(x)`
  Return the cosine of \( x \) radians.

- `math.hypot(x, y)`
  Return the Euclidean norm, \(\sqrt{x^2 + y^2}\). This is the length of the vector from the origin to point \((x, y)\).

- `math.sin(x)`
  Return the sine of \( x \) radians.

- `math.tan(x)`
  Return the tangent of \( x \) radians.
9.2.4. Angular conversion

`math.degrees(x)`
Convert angle x from radians to degrees.

`math.radians(x)`
Convert angle x from degrees to radians.
9.2.5. Hyperbolic functions

math.\texttt{acosh}(x)

Return the inverse hyperbolic cosine of \( x \).

*New in version 2.6.*

math.\texttt{asinh}(x)

Return the inverse hyperbolic sine of \( x \).

*New in version 2.6.*

math.\texttt{atanh}(x)

Return the inverse hyperbolic tangent of \( x \).

*New in version 2.6.*

math.\texttt{cosh}(x)

Return the hyperbolic cosine of \( x \).

math.\texttt{sinh}(x)

Return the hyperbolic sine of \( x \).

math.\texttt{tanh}(x)

Return the hyperbolic tangent of \( x \).
9.2.6. Special functions

- **math.\texttt{erf}(x)**
  Return the error function at \( x \).

  \textit{New in version 2.7.}

- **math.\texttt{erfc}(x)**
  Return the complementary error function at \( x \).

  \textit{New in version 2.7.}

- **math.\texttt{gamma}(x)**
  Return the Gamma function at \( x \).

  \textit{New in version 2.7.}

- **math.\texttt{lgamma}(x)**
  Return the natural logarithm of the absolute value of the Gamma function at \( x \).

  \textit{New in version 2.7.}
9.2.7. Constants

\texttt{math.pi}

The mathematical constant \( \pi = 3.141592... \), to available precision.

\texttt{math.e}

The mathematical constant \( e = 2.718281... \), to available precision.

\textbf{CPython implementation detail:} The \texttt{math} module consists mostly of thin wrappers around the platform C math library functions. Behavior in exceptional cases follows Annex F of the C99 standard where appropriate. The current implementation will raise \texttt{ValueError} for invalid operations like \texttt{sqrt(-1.0)} or \texttt{log(0.0)} (where C99 Annex F recommends signaling invalid operation or divide-by-zero), and \texttt{OverflowError} for results that overflow (for example, \texttt{exp(1000.0)}). A NaN will not be returned from any of the functions above unless one or more of the input arguments was a NaN; in that case, most functions will return a NaN, but (again following C99 Annex F) there are some exceptions to this rule, for example \texttt{pow(float('nan'), 0.0)} or \texttt{hypot(float('nan'), float('inf'))}.

Note that Python makes no effort to distinguish signaling NaNs from quiet NaNs, and behavior for signaling NaNs remains unspecified. Typical behavior is to treat all NaNs as though they were quiet.

\textit{Changed in version 2.6}: Behavior in special cases now aims to follow C99 Annex F. In earlier versions of Python the behavior in special cases was loosely specified.
See also:

**Module `cmath`**

Complex number versions of many of these functions.
This module is always available. It provides access to mathematical functions for complex numbers. The functions in this module accept integers, floating-point numbers or complex numbers as arguments. They will also accept any Python object that has either a __complex__() or a __float__() method: these methods are used to convert the object to a complex or floating-point number, respectively, and the function is then applied to the result of the conversion.

**Note:** On platforms with hardware and system-level support for signed zeros, functions involving branch cuts are continuous on both sides of the branch cut: the sign of the zero distinguishes one side of the branch cut from the other. On platforms that do not support signed zeros the continuity is as specified below.
9.3.1. Conversions to and from polar coordinates

A Python complex number $z$ is stored internally using rectangular or Cartesian coordinates. It is completely determined by its real part $z\.real$ and its imaginary part $z\.imag$. In other words:

\[ z = z\.real + z\.imag*1j \]

Polar coordinates give an alternative way to represent a complex number. In polar coordinates, a complex number $z$ is defined by the modulus $r$ and the phase angle $\phi$. The modulus $r$ is the distance from $z$ to the origin, while the phase $\phi$ is the counterclockwise angle, measured in radians, from the positive $x$-axis to the line segment that joins the origin to $z$.

The following functions can be used to convert from the native rectangular coordinates to polar coordinates and back.

cmath.\texttt{phase}(x)

Return the phase of $x$ (also known as the argument of $x$), as a float. \texttt{phase}(x) is equivalent to math.atan2(x.imag, x.real). The result lies in the range $[-\pi, \pi]$, and the branch cut for this operation lies along the negative real axis, continuous from above. On systems with support for signed zeros (which includes most systems in current use), this means that the sign of the result is the same as the sign of $x\.imag$, even when $x\.imag$ is zero:

\[
>>> \text{phase}(\text{complex}(-1.0, 0.0))
3.1415926535897931
\]
\[
>>> \text{phase}(\text{complex}(-1.0, -0.0))
-3.1415926535897931
\]
New in version 2.6.

Note: The modulus (absolute value) of a complex number \( x \) can be computed using the built-in `abs()` function. There is no separate `cmath` module function for this operation.

`cmath.polar(x)`

Return the representation of \( x \) in polar coordinates. Returns a pair \((r, \phi)\) where \( r \) is the modulus of \( x \) and \( \phi \) is the phase of \( x \). `polar(x)` is equivalent to \((\text{abs}(x), \text{phase}(x))\).

New in version 2.6.

`cmath.rect(r, \phi)`

Return the complex number \( x \) with polar coordinates \( r \) and \( \phi \). Equivalent to \( r * (\text{math.cos}(\phi) + \text{math.sin}(\phi)*1j) \).

New in version 2.6.
9.3.2. Power and logarithmic functions

cmath.\texttt{exp}(x)
Return the exponential value $e^x$.

cmath.\texttt{log}(x[, base])
Returns the logarithm of $x$ to the given \texttt{base}. If the \texttt{base} is not specified, returns the natural logarithm of $x$. There is one branch cut, from 0 along the negative real axis to $-\infty$, continuous from above.

\textit{Changed in version 2.4}: \texttt{base} argument added.

cmath.\texttt{log10}(x)
Return the base-10 logarithm of $x$. This has the same branch cut as \texttt{log()}.

cmath.\texttt{sqrt}(x)
Return the square root of $x$. This has the same branch cut as \texttt{log()}. 
9.3.3. Trigonometric functions

cmath.\texttt{acos}(x)
  Return the arc cosine of $x$. There are two branch cuts: One extends right from 1 along the real axis to $\infty$, continuous from below. The other extends left from -1 along the real axis to -$\infty$, continuous from above.

cmath.\texttt{asin}(x)
  Return the arc sine of $x$. This has the same branch cuts as \texttt{acos()}.

cmath.\texttt{atan}(x)
  Return the arc tangent of $x$. There are two branch cuts: One extends from $1\text{j}$ along the imaginary axis to $\infty\text{j}$, continuous from the right. The other extends from -$1\text{j}$ along the imaginary axis to -$\infty\text{j}$, continuous from the left.

  \textit{Changed in version 2.6:} direction of continuity of upper cut reversed

cmath.\texttt{cos}(x)
  Return the cosine of $x$.

cmath.\texttt{sin}(x)
  Return the sine of $x$.

cmath.\texttt{tan}(x)
  Return the tangent of $x$. 
9.3.4. Hyperbolic functions

```python
cmath.acosh(x)
Return the inverse hyperbolic cosine of x. There is one branch cut, extending left from 1 along the real axis to -∞, continuous from above.
```

```python
cmath.asinh(x)
Return the inverse hyperbolic sine of x. There are two branch cuts: One extends from 1j along the imaginary axis to ∞j, continuous from the right. The other extends from -1j along the imaginary axis to -∞j, continuous from the left.
```

*Changed in version 2.6:* branch cuts moved to match those recommended by the C99 standard

```python
cmath.atanh(x)
Return the inverse hyperbolic tangent of x. There are two branch cuts: One extends from 1 along the real axis to ∞, continuous from below. The other extends from -1 along the real axis to -∞, continuous from above.
```

*Changed in version 2.6:* direction of continuity of right cut reversed

```python
cmath.cosh(x)
Return the hyperbolic cosine of x.
```

```python
cmath.sinh(x)
Return the hyperbolic sine of x.
```

```python
cmath.tanh(x)
Return the hyperbolic tangent of x.
```
9.3.5. Classification functions

`cmath.isinf(x)`

Return `True` if the real or the imaginary part of `x` is positive or negative infinity.

New in version 2.6.

`cmath.isnan(x)`

Return `True` if the real or imaginary part of `x` is not a number (NaN).

New in version 2.6.
9.3.6. Constants

```
cmath.pi
   The mathematical constant \( \pi \), as a float.

cmath.e
   The mathematical constant \( e \), as a float.
```

Note that the selection of functions is similar, but not identical, to that in module `math`. The reason for having two modules is that some users aren’t interested in complex numbers, and perhaps don’t even know what they are. They would rather have `math.sqrt(-1)` raise an exception than return a complex number. Also note that the functions defined in `cmath` always return a complex number, even if the answer can be expressed as a real number (in which case the complex number has an imaginary part of zero).

A note on branch cuts: They are curves along which the given function fails to be continuous. They are a necessary feature of many complex functions. It is assumed that if you need to compute with complex functions, you will understand about branch cuts. Consult almost any (not too elementary) book on complex variables for enlightenment. For information of the proper choice of branch cuts for numerical purposes, a good reference should be the following:

New in version 2.4.

The `decimal` module provides support for decimal floating point arithmetic. It offers several advantages over the `float` datatype:

- Decimal “is based on a floating-point model which was designed with people in mind, and necessarily has a paramount guiding principle – computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school.” – excerpt from the decimal arithmetic specification.

- Decimal numbers can be represented exactly. In contrast, numbers like \(1.1\) and \(2.2\) do not have exact representations in binary floating point. End users typically would not expect \(1.1 + 2.2\) to display as \(3.3000000000000003\) as it does with binary floating point.

- The exactness carries over into arithmetic. In decimal floating point, \(0.1 + 0.1 + 0.1 - 0.3\) is exactly equal to zero. In binary floating point, the result is \(5.5511151231257827\times10^{-17}\). While near to zero, the differences prevent reliable equality testing and differences can accumulate. For this reason, decimal is preferred in accounting applications which have strict equality invariants.

- The decimal module incorporates a notion of significant places so that \(1.30 + 1.20\) is \(2.50\). The trailing zero is kept to indicate significance. This is the customary presentation for monetary applications. For multiplication, the “schoolbook” approach uses all the figures in the multiplicands. For instance, \(1.3 \times 1.2\) gives...
1.56 while 1.30 * 1.20 gives 1.5600.

- Unlike hardware based binary floating point, the decimal module has a user alterable precision (defaulting to 28 places) which can be as large as needed for a given problem:

```python
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- Both binary and decimal floating point are implemented in terms of published standards. While the built-in float type exposes only a modest portion of its capabilities, the decimal module exposes all required parts of the standard. When needed, the programmer has full control over rounding and signal handling. This includes an option to enforce exact arithmetic by using exceptions to block any inexact operations.

- The decimal module was designed to support “without prejudice, both exact unrounded decimal arithmetic (sometimes called fixed-point arithmetic) and rounded floating-point arithmetic.” – excerpt from the decimal arithmetic specification.

The module design is centered around three concepts: the decimal number, the context for arithmetic, and signals.

A decimal number is immutable. It has a sign, coefficient digits, and an exponent. To preserve significance, the coefficient digits do not truncate trailing zeros. Decimals also include special values such as `Infinity`, `-Infinity`, and `NaN`. The standard also differentiates `-0` from `+0`.

The context for arithmetic is an environment specifying precision,
rounding rules, limits on exponents, flags indicating the results of operations, and trap enablers which determine whether signals are treated as exceptions. Rounding options include \texttt{ROUND CEILING}, \texttt{ROUND DOWN}, \texttt{ROUND FLOOR}, \texttt{ROUND HALF DOWN}, \texttt{ROUND HALF EVEN}, \texttt{ROUND HALF UP}, \texttt{ROUND UP}, and \texttt{ROUND 05UP}.

Signals are groups of exceptional conditions arising during the course of computation. Depending on the needs of the application, signals may be ignored, considered as informational, or treated as exceptions. The signals in the decimal module are: \texttt{Clamped}, \texttt{InvalidOperation}, \texttt{DivisionByZero}, \texttt{Inexact}, \texttt{Rounded}, \texttt{Subnormal}, \texttt{Overflow}, and \texttt{Underflow}.

For each signal there is a flag and a trap enabler. When a signal is encountered, its flag is set to one, then, if the trap enabler is set to one, an exception is raised. Flags are sticky, so the user needs to reset them before monitoring a calculation.

\textbf{See also:}

9.4.1. Quick-start Tutorial

The usual start to using decimals is importing the module, viewing the current context with `getcontext()` and, if necessary, setting new values for precision, rounding, or enabled traps:

```python
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999, capitals=1, flags=[], traps=[Overflow, DivisionByZero, InvalidOperation])
```

Decimal instances can be constructed from integers, strings, floats, or tuples. Construction from an integer or a float performs an exact conversion of the value of that integer or float. Decimal numbers include special values such as `NaN` which stands for “Not a number”, positive and negative `Infinity`, and `-0`.

```python
>>> getcontext().prec = 7 # Set a new precision
>>> getcontext().prec = 28
>>> Decimal('10')
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.14')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.41421356237')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.414213562373095048801688724')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```
The significance of a new Decimal is determined solely by the number of digits input. Context precision and rounding only come into play during arithmetic operations.

```python
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

Decimals interact well with much of the rest of Python. Here is a small decimal floating point flying circus:

```python
>>> data = map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split)
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a, b, c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)    # round() first converts to binary floating
1.3
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```
And some mathematical functions are also available to Decimal:

```python
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801688724')
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')
>>> Decimal('10').log10()
Decimal('1')
```

The `quantize()` method rounds a number to a fixed exponent. This method is useful for monetary applications that often round results to a fixed number of places:

```python
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

As shown above, the `getcontext()` function accesses the current context and allows the settings to be changed. This approach meets the needs of most applications.

For more advanced work, it may be useful to create alternate contexts using the `Context()` constructor. To make an alternate active, use the `setcontext()` function.

In accordance with the standard, the `decimal` module provides two ready to use standard contexts, `BasicContext` and `ExtendedContext`. The former is especially useful for debugging because many of the traps are enabled:

```python
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')
```
Contexts also have signal flags for monitoring exceptional conditions encountered during computations. The flags remain set until explicitly cleared, so it is best to clear the flags before each set of monitored computations by using the `clear_flags()` method.

The `flags` entry shows that the rational approximation to \( \pi \) was rounded (digits beyond the context precision were thrown away) and that the result is inexact (some of the discarded digits were non-zero).

Individual traps are set using the dictionary in the `traps` field of a context:
Most programs adjust the current context only once, at the beginning of the program. And, in many applications, data is converted to `Decimal` with a single cast inside a loop. With context set and decimals created, the bulk of the program manipulates the data no differently than with other Python numeric types.
9.4.2. Decimal objects

```python
class decimal.Decimal([value, context])
```

Construct a new `Decimal` object based from `value`.

`value` can be an integer, string, tuple, `float`, or another `Decimal` object. If no `value` is given, returns `Decimal('0')`. If `value` is a string, it should conform to the decimal numeric string syntax after leading and trailing whitespace characters are removed:

```
sign ::= '+' | '-'
digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6'
indicator ::= 'e' | 'E'
digits ::= digit [digit]...
decimal-part ::= digits '.' [digits] | ['.'] digits
exponent-part ::= indicator [sign] digits
infinity ::= 'Infinity' | 'Inf'
nan ::= 'NaN' [digits] | 'sNaN' [digits]
numeric-value ::= decimal-part [exponent-part] | infinity
numeric-string ::= [sign] numeric-value | [sign] nan
```

If `value` is a unicode string then other Unicode decimal digits are also permitted where `digit` appears above. These include decimal digits from various other alphabets (for example, Arabic-Indic and Devanāgarī digits) along with the fullwidth digits u'\uff10' through u'\uff19'.

If `value` is a `tuple`, it should have three components, a sign (0 for positive or 1 for negative), a `tuple` of digits, and an integer exponent. For example, `Decimal((0, (1, 4, 1), -3))` returns `Decimal('1.414')`.

If `value` is a `float`, the binary floating point value is losslessly converted to its exact decimal equivalent. This conversion can
often require 53 or more digits of precision. For example, `Decimal(float('1.1'))` converts to `Decimal('1.1000000000000000888178419700125232338905333447265625')`

The context precision does not affect how many digits are stored. That is determined exclusively by the number of digits in value. For example, `Decimal('3.00000')` records all five zeros even if the context precision is only three.

The purpose of the context argument is determining what to do if value is a malformed string. If the context traps `InvalidOperation`, an exception is raised; otherwise, the constructor returns a new Decimal with the value of NaN.

Once constructed, `Decimal` objects are immutable.

*Changed in version 2.6:* leading and trailing whitespace characters are permitted when creating a Decimal instance from a string.

*Changed in version 2.7:* The argument to the constructor is now permitted to be a `float` instance.

Decimal floating point objects share many properties with the other built-in numeric types such as `float` and `int`. All of the usual math operations and special methods apply. Likewise, decimal objects can be copied, pickled, printed, used as dictionary keys, used as set elements, compared, sorted, and coerced to another type (such as `float` or `long`).

There are some small differences between arithmetic on Decimal objects and arithmetic on integers and floats. When the remainder operator `%` is applied to Decimal objects, the sign of the result is the sign of the dividend rather than the sign of the divisor:
The integer division operator `//` behaves analogously, returning the integer part of the true quotient (truncating towards zero) rather than its floor, so as to preserve the usual identity \( x = (x // y) * y + x \% y \):

```python
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

The `%` and `//` operators implement the `remainder` and `divide-integer` operations (respectively) as described in the specification.

Decimal objects cannot generally be combined with floats in arithmetic operations: an attempt to add a `Decimal` to a `float`, for example, will raise a `TypeError`. There's one exception to this rule: it's possible to use Python's comparison operators to compare a `float` instance \( x \) with a `Decimal` instance \( y \). Without this exception, comparisons between `Decimal` and `float` instances would follow the general rules for comparing objects of different types described in the Expressions section of the reference manual, leading to confusing results.

*Changed in version 2.7:* A comparison between a `float` instance \( x \) and a `Decimal` instance \( y \) now returns a result based on the values of \( x \) and \( y \). In earlier versions \( x < y \) returned the same (arbitrary) result for any `Decimal` instance \( x \) and any `float` instance \( y \).
In addition to the standard numeric properties, decimal floating point objects also have a number of specialized methods:

**adjusted()**
Return the adjusted exponent after shifting out the coefficient’s rightmost digits until only the lead digit remains:
`Decimal('321e+5').adjusted()` returns seven. Used for determining the position of the most significant digit with respect to the decimal point.

**as_tuple()**
Return a *named tuple* representation of the number:
`DecimalTuple(sign, digits, exponent).`  

*Changed in version 2.6:* Use a named tuple.

**canonical()**
Return the canonical encoding of the argument. Currently, the encoding of a *Decimal* instance is always canonical, so this operation returns its argument unchanged.  

*New in version 2.6.*

**compare(other[, context])**
Compare the values of two Decimal instances. This operation behaves in the same way as the usual comparison method `__cmp__()`, except that `compare()` returns a Decimal instance rather than an integer, and if either operand is a NaN then the result is a NaN:

```
a or b is a NaN ==>
Decimal('NaN')
a < b     ==>
Decimal('-1')
a == b     ==>
Decimal('0')
a > b     ==>
Decimal('1')
```
**compare_signal**(other[, context])

This operation is identical to the compare() method, except that all NaNs signal. That is, if neither operand is a signaling NaN then any quiet NaN operand is treated as though it were a signaling NaN.

*New in version 2.6.*

**compare_total**(other)

Compare two operands using their abstract representation rather than their numerical value. Similar to the compare() method, but the result gives a total ordering on Decimal instances. Two Decimal instances with the same numeric value but different representations compare unequal in this ordering:

```python
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

Quiet and signaling NaNs are also included in the total ordering. The result of this function is Decimal('0') if both operands have the same representation, Decimal('-1') if the first operand is lower in the total order than the second, and Decimal('1') if the first operand is higher in the total order than the second operand. See the specification for details of the total order.

*New in version 2.6.*

**compare_total_mag**(other)

Compare two operands using their abstract representation rather than their value as in compare_total(), but ignoring the sign of each operand. x.compare_total_mag(y) is equivalent to x.copy_abs().compare_total(y.copy_abs()).
New in version 2.6.

conjugate()
Just returns self, this method is only to comply with the Decimal Specification.

New in version 2.6.

copy_abs()
Return the absolute value of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

New in version 2.6.

copy_negate()
Return the negation of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

New in version 2.6.

copy_sign(other)
Return a copy of the first operand with the sign set to be the same as the sign of the second operand. For example:

```python
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

New in version 2.6.

exp([context])
Return the value of the (natural) exponential function $e^{**x}$ at
the given number. The result is correctly rounded using the 
**ROUND_HALF_EVEN** rounding mode.

```python
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

*New in version 2.6.*

**`from_float(f)`**

Classmethod that converts a float to a decimal number, exactly.

Note *`Decimal.from_float(0.1)`* is not the same as *`Decimal('0.1')`*. Since 0.1 is not exactly representable in binary floating point, the value is stored as the nearest representable value which is *0x1.999999999999ap-4*. That equivalent value in decimal is *0.10000000000000000555111512312578270211815834045410*.

**Note:** From Python 2.7 onwards, a *`Decimal`* instance can also be constructed directly from a *`float`*.

```python
>>> Decimal.from_float(0.1)
Decimal('0.10000000000000000555111512312578270211815834045410')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

*New in version 2.7.*

**`fma(other, third[, context])`**

Fused multiply-add. Return self*other+third with no rounding
of the intermediate product self*other.

```python
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

*New in version 2.6.*

**is_canonical()**
Return `True` if the argument is canonical and `False` otherwise. Currently, a `Decimal` instance is always canonical, so this operation always returns `True`.

*New in version 2.6.*

**is_finite()**
Return `True` if the argument is a finite number, and `False` if the argument is an infinity or a NaN.

*New in version 2.6.*

**is_infinite()**
Return `True` if the argument is either positive or negative infinity and `False` otherwise.

*New in version 2.6.*

**is_nan()**
Return `True` if the argument is a (quiet or signaling) NaN and `False` otherwise.

*New in version 2.6.*

**is_normal()**
Return `True` if the argument is a `normal` finite non-zero number with an adjusted exponent greater than or equal to
Emin. Return `False` if the argument is zero, subnormal, infinite or a NaN. Note, the term `normal` is used here in a different sense with the `normalize()` method which is used to create canonical values.

*New in version 2.6.*

`is_qnan()`

Return `True` if the argument is a quiet NaN, and `False` otherwise.

*New in version 2.6.*

`is_signed()`

Return `True` if the argument has a negative sign and `False` otherwise. Note that zeros and NaNs can both carry signs.

*New in version 2.6.*

`is_snan()`

Return `True` if the argument is a signaling NaN and `False` otherwise.

*New in version 2.6.*

`is_subnormal()`

Return `True` if the argument is subnormal, and `False` otherwise. A number is subnormal if it is nonzero, finite, and has an adjusted exponent less than `Emin`.

*New in version 2.6.*

`is_zero()`

Return `True` if the argument is a (positive or negative) zero and `False` otherwise.
New in version 2.6.

\texttt{ln([context])}

Return the natural (base e) logarithm of the operand. The result is correctly rounded using the \texttt{ROUND\_HALF\_EVEN} rounding mode.

New in version 2.6.

\texttt{log10([context])}

Return the base ten logarithm of the operand. The result is correctly rounded using the \texttt{ROUND\_HALF\_EVEN} rounding mode.

New in version 2.6.

\texttt{logb([context])}

For a nonzero number, return the adjusted exponent of its operand as a \texttt{Decimal} instance. If the operand is a zero then \texttt{Decimal('-Infinity')} is returned and the \texttt{DivisionByZero} flag is raised. If the operand is an infinity then \texttt{Decimal('Infinity')} is returned.

New in version 2.6.

\texttt{logical\_and(other[, context])}

\texttt{logical\_and()} is a logical operation which takes two \texttt{logical operands} (see Logical operands). The result is the digit-wise and of the two operands.

New in version 2.6.

\texttt{logical\_invert([context])}

\texttt{logical\_invert()} is a logical operation. The result is the digit-
wise inversion of the operand.

*New in version 2.6.*

```
logical_or(other[, context])
logical_or() is a logical operation which takes two logical operands (see Logical operands). The result is the digit-wise or of the two operands.

*New in version 2.6.*
```

```
logical_xor(other[, context])
logical_xor() is a logical operation which takes two logical operands (see Logical operands). The result is the digit-wise exclusive or of the two operands.

*New in version 2.6.*
```

```
max(other[, context])
Like max(self, other) except that the context rounding rule is applied before returning and that NaN values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

max_mag(other[, context])
Similar to the max() method, but the comparison is done using the absolute values of the operands.

*New in version 2.6.*
```

```
min(other[, context])
Like min(self, other) except that the context rounding rule is applied before returning and that NaN values are either
signaled or ignored (depending on the context and whether they are signaling or quiet).

\texttt{min\_mag(\textit{other[, context]})}

Similar to the \texttt{min()} method, but the comparison is done using the absolute values of the operands.

\textit{New in version 2.6.}

\texttt{next\_minus(\textit{context])}

Return the largest number representable in the given context (or in the current thread's context if no context is given) that is smaller than the given operand.

\textit{New in version 2.6.}

\texttt{next\_plus(\textit{context])}

Return the smallest number representable in the given context (or in the current thread's context if no context is given) that is larger than the given operand.

\textit{New in version 2.6.}

\texttt{next\_toward(\textit{other[, context]})}

If the two operands are unequal, return the number closest to the first operand in the direction of the second operand. If both operands are numerically equal, return a copy of the first operand with the sign set to be the same as the sign of the second operand.

\textit{New in version 2.6.}

\texttt{normalize(\textit{context])}

Normalize the number by stripping the rightmost trailing zeros
and converting any result equal to `Decimal('0')` to
`Decimal('0e0')`. Used for producing canonical values for
attributes of an equivalence class. For example,
`Decimal('32.100')` and `Decimal('0.321000e+2')` both
normalize to the equivalent value `Decimal('32.1')`.

`number_class([context])`

Return a string describing the class of the operand. The
returned value is one of the following ten strings.

- "-Infinity", indicating that the operand is negative
  infinity.
- "-Normal", indicating that the operand is a negative
  normal number.
- "-Subnormal", indicating that the operand is negative and
  subnormal.
- "-Zero", indicating that the operand is a negative zero.
- "+Zero", indicating that the operand is a positive zero.
- "+Subnormal", indicating that the operand is positive and
  subnormal.
- "+Normal", indicating that the operand is a positive
  normal number.
- "+Infinity", indicating that the operand is positive
  infinity.
- "NaN", indicating that the operand is a quiet NaN (Not a
  Number).
- "sNaN", indicating that the operand is a signaling NaN.

*New in version 2.6.*

`quantize(exp[, rounding[, context[, watchexp]]])`

Return a value equal to the first operand after rounding and
having the exponent of the second operand.
Unlike other operations, if the length of the coefficient after the quantize operation would be greater than precision, then an InvalidOperation is signaled. This guarantees that, unless there is an error condition, the quantized exponent is always equal to that of the right-hand operand.

Also unlike other operations, quantize never signals Underflow, even if the result is subnormal and inexact.

If the exponent of the second operand is larger than that of the first then rounding may be necessary. In this case, the rounding mode is determined by the rounding argument if given, else by the given context argument; if neither argument is given the rounding mode of the current thread’s context is used.

If watchexp is set (default), then an error is returned whenever the resulting exponent is greater than Emax or less than Etiny.

radix()
Return Decimal(10), the radix (base) in which the Decimal class does all its arithmetic. Included for compatibility with the specification.

New in version 2.6.

remainder_near(other[, context])
Return the remainder from dividing self by other. This differs from self % other in that the sign of the remainder is chosen so as to minimize its absolute value. More precisely, the return value is self - n * other where n is the integer nearest to the exact value of self / other, and if two integers
are equally near then the even one is chosen.

If the result is zero then its sign will be the sign of self.

```python
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

**rotate**(other[, context])

Return the result of rotating the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to rotate. If the second operand is positive then rotation is to the left; otherwise rotation is to the right. The coefficient of the first operand is padded on the left with zeros to length precision if necessary. The sign and exponent of the first operand are unchanged.

*New in version 2.6.*

**same_quantum**(other[, context])

Test whether self and other have the same exponent or whether both are NaN.

**scaleb**(other[, context])

Return the first operand with exponent adjusted by the second. Equivalently, return the first operand multiplied by \(10^{**other}\). The second operand must be an integer.

*New in version 2.6.*

**shift**(other[, context])
Return the result of shifting the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range \(-\text{precision}\) through \(\text{precision}\). The absolute value of the second operand gives the number of places to shift. If the second operand is positive then the shift is to the left; otherwise the shift is to the right. Digits shifted into the coefficient are zeros. The sign and exponent of the first operand are unchanged.

*New in version 2.6.*

\[\text{sqrt}([\text{context}])\]
Return the square root of the argument to full precision.

\[\text{to\_eng\_string}([\text{context}])\]
Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

For example, this converts \(\text{Decimal('123E+1')}\) to \(\text{Decimal('1.23E+3')}\).

\[\text{to\_integral}([\text{rounding}[\], \text{context}][])\]
Identical to the \text{to\_integral\_value()} method. The \text{to\_integral} name has been kept for compatibility with older versions.

\[\text{to\_integral\_exact}([\text{rounding}[\], \text{context}][])\]
Round to the nearest integer, signaling **Inexact** or **Rounded** as appropriate if rounding occurs. The rounding mode is
determined by the \texttt{rounding} parameter if given, else by the given \texttt{context}. If neither parameter is given then the rounding mode of the current context is used.

\textit{New in version 2.6.}

\texttt{to_integral_value}([\texttt{rounding}[\texttt{context}])

Round to the nearest integer without signaling \texttt{Inexact} or \texttt{Rounded}. If given, applies \texttt{rounding}; otherwise, uses the rounding method in either the supplied \texttt{context} or the current context.

\textit{Changed in version 2.6:} renamed from \texttt{to_integral} to \texttt{to_integral_value}. The old name remains valid for compatibility.

\textbf{9.4.2.1. Logical operands}

The \texttt{logical_and()}, \texttt{logical_invert()}, \texttt{logical_or()}, and \texttt{logical_xor()} methods expect their arguments to be \textit{logical operands}. A \textit{logical operand} is a \texttt{Decimal} instance whose exponent and sign are both zero, and whose digits are all either \texttt{0} or \texttt{1}.
9.4.3. Context objects

Contexts are environments for arithmetic operations. They govern precision, set rules for rounding, determine which signals are treated as exceptions, and limit the range for exponents.

Each thread has its own current context which is accessed or changed using the `getcontext()` and `setcontext()` functions:

```python
decimal.getcontext()
    Return the current context for the active thread.

decimal.setcontext(c)
    Set the current context for the active thread to c.
```

Beginning with Python 2.5, you can also use the `with` statement and the `localcontext()` function to temporarily change the active context.

```python
decimal.localcontext([c])
    Return a context manager that will set the current context for the active thread to a copy of c on entry to the with-statement and restore the previous context when exiting the with-statement. If no context is specified, a copy of the current context is used.
```

*New in version 2.5.*

For example, the following code sets the current decimal precision to 42 places, performs a calculation, and then automatically restores the previous context:

```python
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42          # Perform a high precision calculation
s = calculate_something()
```
New contexts can also be created using the `Context` constructor described below. In addition, the module provides three pre-made contexts:

```python
s = +s  # Round the final result back to the default precision

with localcontext(BasicContext):  # temporarily use the
    print Decimal(1) / Decimal(7)
    print Decimal(355) / Decimal(113)
```

### `class decimal.BasicContext`

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to `ROUND_HALF_UP`. All flags are cleared. All traps are enabled (treated as exceptions) except `Inexact`, `Rounded`, and `Subnormal`.

Because many of the traps are enabled, this context is useful for debugging.

### `class decimal.ExtendedContext`

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to `ROUND_HALF_EVEN`. All flags are cleared. No traps are enabled (so that exceptions are not raised during computations).

Because the traps are disabled, this context is useful for applications that prefer to have result value of `NaN` or `Infinity` instead of raising exceptions. This allows an application to complete a run in the presence of conditions that would otherwise halt the program.

### `class decimal.DefaultContext`

This context is used by the `Context` constructor as a prototype for new contexts. Changing a field (such a precision) has the effect of changing the default for new contexts created by the `Context`
class decimal.\texttt{Context}(prec=None, rounding=None, traps=None, flags=None, Emin=None, Emax=None, capitals=1)

    Creates a new context. If a field is not specified or is \texttt{None}, the default values are copied from the \texttt{DefaultContext}. If the \texttt{flags} field is not specified or is \texttt{None}, all flags are cleared.

    The \texttt{prec} field is a positive integer that sets the precision for arithmetic operations in the context.

    The \texttt{rounding} option is one of:

    - \texttt{ROUND\_CEILING} (towards Infinity),
    - \texttt{ROUND\_DOWN} (towards zero),
    - \texttt{ROUND\_FLOOR} (towards -Infinity),
    - \texttt{ROUND\_HALF\_DOWN} (to nearest with ties going towards zero),
    - \texttt{ROUND\_HALF\_EVEN} (to nearest with ties going to nearest even
integer),
- **ROUND_HALF_UP** (to nearest with ties going away from zero), or
- **ROUND_UP** (away from zero).
- **ROUND_HALF_UP** (away from zero if last digit after rounding towards zero would have been 0 or 5; otherwise towards zero)

The *traps* and *flags* fields list any signals to be set. Generally, new contexts should only set traps and leave the flags clear.

The *Emin* and *Emax* fields are integers specifying the outer limits allowable for exponents.

The *capitals* field is either 0 or 1 (the default). If set to 1, exponents are printed with a capital E; otherwise, a lowercase e is used: `Decimal('6.02e+23')`.

*Changed in version 2.6:* The **ROUND_HALF_UP** rounding mode was added.

The **Context** class defines several general purpose methods as well as a large number of methods for doing arithmetic directly in a given context. In addition, for each of the **Decimal** methods described above (with the exception of the adjusted() and as_tuple() methods) there is a corresponding **Context** method. For example, for a **Context** instance `C` and **Decimal** instance `x`, `C.exp(x)` is equivalent to `x.exp(context=C)`. Each **Context** method accepts a Python integer (an instance of `int` or `long`) anywhere that a Decimal instance is accepted.

**clear_flags()**

Resets all of the flags to 0.

**copy()**
Return a duplicate of the context.

**copy_decimal(num)**

Return a copy of the Decimal instance num.

**create_decimal(num)**

Creates a new Decimal instance from num but using self as context. Unlike the Decimal constructor, the context precision, rounding method, flags, and traps are applied to the conversion.

This is useful because constants are often given to a greater precision than is needed by the application. Another benefit is that rounding immediately eliminates unintended effects from digits beyond the current precision. In the following example, using unrounded inputs means that adding zero to a sum can change the result:

```python
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

This method implements the to-number operation of the IBM specification. If the argument is a string, no leading or trailing whitespace is permitted.

**create_decimal_from_float(f)**

Creates a new Decimal instance from a float f but rounding using self as the context. Unlike the Decimal.from_float() class method, the context precision, rounding method, flags, and traps are applied to the conversion.

```python
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
```
New in version 2.7.

Etiny()

Returns a value equal to $E_{\text{min}} - \text{prec} + 1$ which is the minimum exponent value for subnormal results. When underflow occurs, the exponent is set to Etiny.

Etop()

Returns a value equal to $E_{\text{max}} - \text{prec} + 1$.

The usual approach to working with decimals is to create Decimal instances and then apply arithmetic operations which take place within the current context for the active thread. An alternative approach is to use context methods for calculating within a specific context. The methods are similar to those for the Decimal class and are only briefly recounted here.

abs($x$)

Returns the absolute value of $x$.

add($x$, $y$)

Return the sum of $x$ and $y$.

canonical($x$)

Returns the same Decimal object $x$.

compare($x$, $y$)

Compares $x$ and $y$ numerically.

compare_signal($x$, $y$)
Compares the values of the two operands numerically.

**compare_total**(*x*, *y*)

Compares two operands using their abstract representation.

**compare_total_mag**(*x*, *y*)

Compares two operands using their abstract representation, ignoring sign.

**copy_abs**(*x*)

Returns a copy of *x* with the sign set to 0.

**copy_negate**(*x*)

Returns a copy of *x* with the sign inverted.

**copy_sign**(*x*, *y*)

Copies the sign from *y* to *x*.

**divide**(*x*, *y*)

Return *x* divided by *y*.

**divide_int**(*x*, *y*)

Return *x* divided by *y*, truncated to an integer.

**divmod**(*x*, *y*)

Divides two numbers and returns the integer part of the result.

**exp**(*x*)

Returns e ** *x*.

**fma**(*x*, *y*, *z*)

Returns *x* multiplied by *y*, plus *z*.

**is_canonical**(*x*)

Returns True if *x* is canonical; otherwise returns False.
is_finite(x)
    Returns True if x is finite; otherwise returns False.

is_infinite(x)
    Returns True if x is infinite; otherwise returns False.

is_nan(x)
    Returns True if x is a qNaN or sNaN; otherwise returns False.

is_normal(x)
    Returns True if x is a normal number; otherwise returns False.

is_qnan(x)
    Returns True if x is a quiet NaN; otherwise returns False.

is_signed(x)
    Returns True if x is negative; otherwise returns False.

is_snan(x)
    Returns True if x is a signaling NaN; otherwise returns False.

is_subnormal(x)
    Returns True if x is subnormal; otherwise returns False.

is_zero(x)
    Returns True if x is a zero; otherwise returns False.

ln(x)
    Returns the natural (base e) logarithm of x.

log10(x)
    Returns the base 10 logarithm of x.

logb(x)
Returns the exponent of the magnitude of the operand's MSD.

`logical_and(x, y)`
Applies the logical operation `and` between each operand's digits.

`logical_invert(x)`
Invert all the digits in `x`.

`logical_or(x, y)`
Applies the logical operation `or` between each operand's digits.

`logical_xor(x, y)`
Applies the logical operation `xor` between each operand's digits.

`max(x, y)`
Compares two values numerically and returns the maximum.

`max_mag(x, y)`
Compares the values numerically with their sign ignored.

`min(x, y)`
Compares two values numerically and returns the minimum.

`min_mag(x, y)`
Compares the values numerically with their sign ignored.

`minus(x)`
Minus corresponds to the unary prefix minus operator in Python.

`multiply(x, y)`
Return the product of `x` and `y`. 
next_minus(x)
Returns the largest representable number smaller than x.

next_plus(x)
Returns the smallest representable number larger than x.

next_toward(x, y)
Returns the number closest to x, in direction towards y.

normalize(x)
Reduces x to its simplest form.

number_class(x)
Returns an indication of the class of x.

plus(x)
Plus corresponds to the unary prefix plus operator in Python.
This operation applies the context precision and rounding, so
it is not an identity operation.

power(x, y[, modulo])
Return x to the power of y, reduced modulo modulo if given.

With two arguments, compute \(x^{*\ y}\). If x is negative then y
must be integral. The result will be inexact unless y is integral
and the result is finite and can be expressed exactly in
‘precision’ digits. The result should always be correctly
rounded, using the rounding mode of the current thread’s
context.

With three arguments, compute \((x^{*\ y}) \ % \ modulo\). For the
three argument form, the following restrictions on the
arguments hold:

- all three arguments must be integral
- \( y \) must be nonnegative
- at least one of \( x \) or \( y \) must be nonzero
- \texttt{modulo} must be nonzero and have at most ‘precision’ digits

The value resulting from \( \text{Context.power}(x, y, \text{modulo}) \) is equal to the value that would be obtained by computing \( (x**y) \% \text{modulo} \) with unbounded precision, but is computed more efficiently. The exponent of the result is zero, regardless of the exponents of \( x, y \) and \texttt{modulo}. The result is always exact.

*Changed in version 2.6:* \( y \) may now be nonintegral in \( x**y \). Stricter requirements for the three-argument version.

**quantize**(\( x, y \))
- Returns a value equal to \( x \) (rounded), having the exponent of \( y \).

**radix**()
- Just returns 10, as this is Decimal, :)

**remainder**(\( x, y \))
- Returns the remainder from integer division.

  The sign of the result, if non-zero, is the same as that of the original dividend.

**remainder_near**(\( x, y \))
- Returns \( x - y * n \), where \( n \) is the integer nearest the exact value of \( x / y \) (if the result is 0 then its sign will be the sign of \( x \)).

**rotate**(\( x, y \))
Returns a rotated copy of $x$, $y$ times.

**same_quantum**($x$, $y$)
Returns True if the two operands have the same exponent.

**scaleb**($x$, $y$)
Returns the first operand after adding the second value its exp.

**shift**($x$, $y$)
Returns a shifted copy of $x$, $y$ times.

**sqrt**($x$)
Square root of a non-negative number to context precision.

**subtract**($x$, $y$)
Return the difference between $x$ and $y$.

**to_eng_string**($x$)
Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

**to_integral_exact**($x$)
Rounds to an integer.

**to_sci_string**($x$)
Converts a number to a string using scientific notation.
9.4.4. Signals

Signals represent conditions that arise during computation. Each corresponds to one context flag and one context trap enabler.

The context flag is set whenever the condition is encountered. After the computation, flags may be checked for informational purposes (for instance, to determine whether a computation was exact). After checking the flags, be sure to clear all flags before starting the next computation.

If the context's trap enabler is set for the signal, then the condition causes a Python exception to be raised. For example, if the DivisionByZero trap is set, then a DivisionByZero exception is raised upon encountering the condition.

```python
class decimal.Clamped
    Altered an exponent to fit representation constraints.

    Typically, clamping occurs when an exponent falls outside the context's Emin and Emax limits. If possible, the exponent is reduced to fit by adding zeros to the coefficient.

class decimal.DecimalException
    Base class for other signals and a subclass of ArithmeticError.

class decimal.DivisionByZero
    Signals the division of a non-infinite number by zero.

    Can occur with division, modulo division, or when raising a number to a negative power. If this signal is not trapped, returns Infinity or -Infinity with the sign determined by the inputs to the calculation.
```
class decimal. **Inexact**
Indicates that rounding occurred and the result is not exact.

Signals when non-zero digits were discarded during rounding. The rounded result is returned. The signal flag or trap is used to detect when results are inexact.

class decimal. **InvalidOperation**
An invalid operation was performed.

Indicates that an operation was requested that does not make sense. If not trapped, returns NaN. Possible causes include:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
x._rescale( non-integer )
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

class decimal. **Overflow**
Numerical overflow.

Indicates the exponent is larger than Emax after rounding has occurred. If not trapped, the result depends on the rounding mode, either pulling inward to the largest representable finite number or rounding outward to Infinity. In either case, Inexact and Rounded are also signaled.

class decimal. **Rounded**
Rounding occurred though possibly no information was lost.

Signaled whenever rounding discards digits; even if those digits
are zero (such as rounding \(5.00\) to \(5.0\)). If not trapped, returns the result unchanged. This signal is used to detect loss of significant digits.

*class* decimal. **Subnormal**

Exponent was lower than \(E_{\text{min}}\) prior to rounding.

Occurs when an operation result is subnormal (the exponent is too small). If not trapped, returns the result unchanged.

*class* decimal. **Underflow**

Numerical underflow with result rounded to zero.

Occurs when a subnormal result is pushed to zero by rounding. **Inexact** and **Subnormal** are also signaled.

The following table summarizes the hierarchy of signals:

```python
exceptions.ArithmeticError(exceptions.StandardError)
DecimalException
  Clamped
DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
Inexact
  Overflow(Inexact, Rounded)
  Underflow(Inexact, Rounded, Subnormal)
InvalidOperation
Rounded
Subnormal
```
9.4.5. Floating Point Notes

9.4.5.1. Mitigating round-off error with increased precision

The use of decimal floating point eliminates decimal representation error (making it possible to represent \(0.1\) exactly); however, some operations can still incur round-off error when non-zero digits exceed the fixed precision.

The effects of round-off error can be amplified by the addition or subtraction of nearly offsetting quantities resulting in loss of significance. Knuth provides two instructive examples where rounded floating point arithmetic with insufficient precision causes the breakdown of the associative and distributive properties of addition:

```python
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(1111113), Decimal(-1111111), Decimal('7
   Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

The `decimal` module makes it possible to restore the identities by expanding the precision sufficiently to avoid loss of significance:
9.4.5.2. Special values

The number system for the `decimal` module provides special values including `NaN`, `sNaN`, `-Infinity`, `Infinity`, and two zeros, `+0` and `-0`.

Infinities can be constructed directly with: `Decimal('Infinity')`. Also, they can arise from dividing by zero when the `DivisionByZero` signal is not trapped. Likewise, when the `Overflow` signal is not trapped, infinity can result from rounding beyond the limits of the largest representable number.

The infinities are signed (affine) and can be used in arithmetic operations where they get treated as very large, indeterminate numbers. For instance, adding a constant to infinity gives another infinite result.

Some operations are indeterminate and return `NaN`, or if the `InvalidOperation` signal is trapped, raise an exception. For example, `0/0` returns `NaN` which means “not a number”. This variety of `NaN` is quiet and, once created, will flow through other computations always resulting in another `NaN`. This behavior can be useful for a series of computations that occasionally have missing inputs — it allows the calculation to proceed while flagging specific results as invalid.
A variant is \texttt{sNaN} which signals rather than remaining quiet after every operation. This is a useful return value when an invalid result needs to interrupt a calculation for special handling.

The behavior of Python’s comparison operators can be a little surprising where a \texttt{NaN} is involved. A test for equality where one of the operands is a quiet or signaling \texttt{NaN} always returns \texttt{False} (even when doing \texttt{Decimal('NaN') == Decimal('NaN')}), while a test for inequality always returns \texttt{True}. An attempt to compare two Decimals using any of the \texttt{<, <=, >} or \texttt{>=} operators will raise the \texttt{InvalidOperation} signal if either operand is a \texttt{NaN}, and return \texttt{False} if this signal is not trapped. Note that the General Decimal Arithmetic specification does not specify the behavior of direct comparisons; these rules for comparisons involving a \texttt{NaN} were taken from the IEEE 854 standard (see Table 3 in section 5.7). To ensure strict standards-compliance, use the \texttt{compare()} and \texttt{compare-signal()} methods instead.

The signed zeros can result from calculations that underflow. They keep the sign that would have resulted if the calculation had been carried out to greater precision. Since their magnitude is zero, both positive and negative zeros are treated as equal and their sign is informational.

In addition to the two signed zeros which are distinct yet equal, there are various representations of zero with differing precisions yet equivalent in value. This takes a bit of getting used to. For an eye accustomed to normalized floating point representations, it is not immediately obvious that the following calculation returns a value equal to zero:

```python
>>> 1 / Decimal('Infinity')
Decimal('0E-1000000026')
```
9.4.6. Working with threads

The `getcontext()` function accesses a different `Context` object for each thread. Having separate thread contexts means that threads may make changes (such as `getcontext.prec=10`) without interfering with other threads.

Likewise, the `setcontext()` function automatically assigns its target to the current thread.

If `setcontext()` has not been called before `getcontext()`, then `getcontext()` will automatically create a new context for use in the current thread.

The new context is copied from a prototype context called `DefaultContext`. To control the defaults so that each thread will use the same values throughout the application, directly modify the `DefaultContext` object. This should be done before any threads are started so that there won’t be a race condition between threads calling `getcontext()`.

For example:

```python
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started

t1.start()
t2.start()
t3.start()
...```

```
9.4.7. Recipes

Here are a few recipes that serve as utility functions and that demonstrate ways to work with the `Decimal` class:

```python
def moneyfmt(value, places=2, curr='', sep=':', dp='.', pos='', neg='-', trailneg=''):  
    """Convert Decimal to a money formatted string.  
places: required number of places after the decimal point  
curr: optional currency symbol before the sign (may be blank)  
sep: optional grouping separator (comma, period, space)  
dp: decimal point indicator (comma or period)  
only specify as blank when places is zero  
pos: optional sign for positive numbers: '+', space or blank  
neg: optional sign for negative numbers: '-', '(', space etc  
trailneg:optional trailing minus indicator: '-', ')', space etc  
""
    q = Decimal(10) ** -places  # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = map(str, digits)
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        build(next() if digits else '0')
    build(dp)
    if not digits:
        build('0')
```

```shell
>>> d = Decimal('-1234567.8901')
>>> moneyfmt(d, curr='$')
'-$1,234,567.89'
>>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
'1.234.568-'
>>> moneyfmt(d, curr='$', neg='(', trailneg=')')
'($1,234,567.89)'
>>> moneyfmt(Decimal(123456789), sep='	')
'123	456	789.00'
>>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
'<0.02>'
```

```python
q = Decimal(10) ** -places  # 2 places --> '0.01'
```
i = 0
while digits:
    build(next())
    i += 1
    if i == 3 and digits:
        i = 0
        build(sep)
    build(curr)
    build(neg if sign else pos)
return '' . join(reversed(result))

def pi():
    """Compute Pi to the current precision."

    >>> print pi()
    3.141592653589793238462643383

    """
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)  # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2  # unary plus applies the new precision
    return +s

def exp(x):
    """Return e raised to the power of x. Result type matches input type."

    >>> print exp(Decimal(1))
    2.718281828459045235360287471
    >>> print exp(Decimal(2))
    7.389056098930650227230427461
    >>> print exp(2.0)
    7.38905609893
    >>> print exp(2+0j)
    (7.38905609893+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
def cos(x):
    
    return +s

def sin(x):
    
    return +s
num *= x * x
sign *= -1
s += num / fact * sign
getcontext().prec -= 2
return +s
9.4.8. Decimal FAQ

Q. It is cumbersome to type `decimal.Decimal('1234.5')`. Is there a way to minimize typing when using the interactive interpreter?

A. Some users abbreviate the constructor to just a single letter:

```python
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. In a fixed-point application with two decimal places, some inputs have many places and need to be rounded. Others are not supposed to have excess digits and need to be validated. What methods should be used?

A. The `quantize()` method rounds to a fixed number of decimal places. If the `Inexact` trap is set, it is also useful for validation:

```python
>>> TWOPLACES = Decimal(10) ** -2  # same as Decimal('0.01')

>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')

>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')

>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
...
Inexact: None
```

Q. Once I have valid two place inputs, how do I maintain that
invariant throughout an application?

A. Some operations like addition, subtraction, and multiplication by an integer will automatically preserve fixed point. Others operations, like division and non-integer multiplication, will change the number of decimal places and need to be followed-up with a `quantize()` step:

```python
>>> a = Decimal('102.72') # Initial fixed-point value
>>> b = Decimal('3.17')
>>> a + b
Decimal('105.89') # Addition preserves fixed-point
>>> a - b
Decimal('99.55')
>>> a * 42
Decimal('4314.24') # So does integer multiplication
>>> (a * b).quantize(TWOPLACES) # Must quantize non-integer
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES) # And quantize division
Decimal('0.03')
```

In developing fixed-point applications, it is convenient to define functions to handle the `quantize()` step:

```python
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)
```

```python
>>> mul(a, b) # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

Q. There are many ways to express the same value. The numbers `200`, `200.000`, `2E2`, and `02E+4` all have the same value at various precisions. Is there a way to transform them to a single recognizable canonical value?
A. The `normalize()` method maps all equivalent values to a single representative:

```python
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. Some decimal values always print with exponential notation. Is there a way to get a non-exponential representation?

A. For some values, exponential notation is the only way to express the number of significant places in the coefficient. For example, expressing `5.0E+3` as `5000` keeps the value constant but cannot show the original’s two-place significance.

If an application does not care about tracking significance, it is easy to remove the exponent and trailing zeros, losing significance, but keeping the value unchanged:

```python
def remove_exponent(d):
    '''Remove exponent and trailing zeros.'''
    return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

Q. Is there a way to convert a regular float to a `Decimal`?

A. Yes, any binary floating point number can be exactly expressed as a Decimal though an exact conversion may take more precision than intuition would suggest:

```python
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```
Q. Within a complex calculation, how can I make sure that I haven’t gotten a spurious result because of insufficient precision or rounding anomalies.

A. The decimal module makes it easy to test results. A best practice is to re-run calculations using greater precision and with various rounding modes. Widely differing results indicate insufficient precision, rounding mode issues, ill-conditioned inputs, or a numerically unstable algorithm.

Q. I noticed that context precision is applied to the results of operations but not to the inputs. Is there anything to watch out for when mixing values of different precisions?

A. Yes. The principle is that all values are considered to be exact and so is the arithmetic on those values. Only the results are rounded. The advantage for inputs is that “what you type is what you get”. A disadvantage is that the results can look odd if you forget that the inputs haven’t been rounded:

```python
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

The solution is either to increase precision or to force rounding of inputs using the unary plus operation:

```python
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')  # unary plus triggers rounding
Decimal('1.23')
```

Alternatively, inputs can be rounded upon creation using the `Context.create_decimal()` method:

```python
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345
```
Decimal('1.2345')
9.5. fractions — Rational numbers

New in version 2.6.

Source code: Lib/fractions.py

The fractions module provides support for rational number arithmetic.

A Fraction instance can be constructed from a pair of integers, from another rational number, or from a string.

```python
class fractions.Fraction(numerator=0, denominator=1)
class fractions.Fraction(other_fraction)
class fractions.Fraction(float)
class fractions.Fraction(decimal)
class fractions.Fraction(string)
```

The first version requires that numerator and denominator are instances of numbers.Rational and returns a new Fraction instance with value numerator/denominator. If denominator is 0, it raises a ZeroDivisionError. The second version requires that other_fraction is an instance of numbers.Rational and returns a Fraction instance with the same value. The next two versions accept either a float or a decimal.Decimal instance, and return a Fraction instance with exactly the same value. Note that due to the usual issues with binary floating-point (see Floating Point Arithmetic: Issues and Limitations), the argument to Fraction(1.1) is not exactly equal to 11/10, and so Fraction(1.1) does not return Fraction(11, 10) as one might expect. (But see the documentation for the limit_denominator() method below.) The last version of the constructor expects a string or unicode instance. The usual form for this instance is:
[sign] numerator ['/' denominator]

where the optional sign may be either `+` or `-` and numerator and denominator (if present) are strings of decimal digits. In addition, any string that represents a finite value and is accepted by the float constructor is also accepted by the Fraction constructor. In either form the input string may also have leading and/or trailing whitespace. Here are some examples:

```python
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

The Fraction class inherits from the abstract base class numbers.Rational, and implements all of the methods and operations from that class. Fraction instances are hashable, and should be treated as immutable. In addition, Fraction has the following methods:
Changed in version 2.7: The `Fraction` constructor now accepts `float` and `decimal.Decimal` instances.

`from_float(flt)`

This class method constructs a `Fraction` representing the exact value of `flt`, which must be a `float`. Beware that `Fraction.from_float(0.3)` is not the same value as `Fraction(3, 10).

**Note:** From Python 2.7 onwards, you can also construct a `Fraction` instance directly from a `float`.

`from_decimal(dec)`

This class method constructs a `Fraction` representing the exact value of `dec`, which must be a `decimal.Decimal`.

**Note:** From Python 2.7 onwards, you can also construct a `Fraction` instance directly from a `decimal.Decimal` instance.

`limit_denominator(max_denominator=1000000)`

Finds and returns the closest `Fraction` to `self` that has denominator at most `max_denominator`. This method is useful for finding rational approximations to a given floating-point number:

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

or for recovering a rational number that's represented as a float:

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
```
fractions. **gcd**\((a, b)\)

Return the greatest common divisor of the integers \(a\) and \(b\). If either \(a\) or \(b\) is nonzero, then the absolute value of \(\text{gcd}(a, b)\) is the largest integer that divides both \(a\) and \(b\). \(\text{gcd}(a, b)\) has the same sign as \(b\) if \(b\) is nonzero; otherwise it takes the sign of \(a\). \(\text{gcd}(0, 0)\) returns 0.

**See also:**

**Module** numbers

The abstract base classes making up the numeric tower.
This module implements pseudo-random number generators for various distributions.

For integers, uniform selection from a range. For sequences, uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

On the real line, there are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distributions of angles, the von Mises distribution is available.

Almost all module functions depend on the basic function `random()`, which generates a random float uniformly in the semi-open range [0.0, 1.0). Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of $2^{19937} - 1$. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

The functions supplied by this module are actually bound methods of a hidden instance of the `random.Random` class. You can instantiate your own instances of `Random` to get generators that don’t share state. This is especially useful for multi-threaded programs, creating
a different instance of Random for each thread, and using the 
jumpahead() method to make it likely that the generated sequences 
seen by each thread don't overlap.

Class Random can also be subclassed if you want to use a different 
basic generator of your own devising: in that case, override the 
random(), seed(), getstate(), setstate() and jumpahead() methods. 
Optionally, a new generator can supply a getrandbits() method — 
this allows randrange() to produce selections over an arbitrarily large 
range.

New in version 2.4: the getrandbits() method.

As an example of subclassing, the random module provides the 
WichmannHill class that implements an alternative generator in pure 
Python. The class provides a backward compatible way to reproduce 
results from earlier versions of Python, which used the Wichmann-
Hill algorithm as the core generator. Note that this Wichmann-Hill 
generator can no longer be recommended: its period is too short by 
contemporary standards, and the sequence generated is known to 
fail some stringent randomness tests. See the references below for a 
recent variant that repairs these flaws.

Changed in version 2.3: MersenneTwister replaced Wichmann-Hill 
as the default generator.

The random module also provides the SystemRandom class which uses 
the system function os.urandom() to generate random numbers from 
sources provided by the operating system.

**Warning:** The pseudo-random generators of this module should 
not be used for security purposes. Use os.urandom() or 
SystemRandom if you require a cryptographically secure pseudo- 
random number generator.
Bookkeeping functions:

`random.seed([x])`

Initialize the basic random number generator. Optional argument `x` can be any hashable object. If `x` is omitted or `None`, current system time is used; current system time is also used to initialize the generator when the module is first imported. If randomness sources are provided by the operating system, they are used instead of the system time (see the `os.urandom()` function for details on availability).

If a hashable object is given, deterministic results are only assured when `PYTHONHASHSEED` is disabled.

*Changed in version 2.4*: formerly, operating system resources were not used.

`random.getstate()`

Return an object capturing the current internal state of the generator. This object can be passed to `setstate()` to restore the state.

*New in version 2.1.*

*Changed in version 2.6*: State values produced in Python 2.6 cannot be loaded into earlier versions.

`random.setstate(state)`

`state` should have been obtained from a previous call to `getstate()`, and `setstate()` restores the internal state of the generator to what it was at the time `getstate()` was called.

*New in version 2.1.*
random. `jumpahead(n)`

Change the internal state to one different from and likely far away from the current state. *n* is a non-negative integer which is used to scramble the current state vector. This is most useful in multi-threaded programs, in conjunction with multiple instances of the `Random` class: `setstate()` or `seed()` can be used to force all instances into the same internal state, and then `jumpahead()` can be used to force the instances’ states far apart.

*New in version 2.1.*

*Changed in version 2.3:* Instead of jumping to a specific state, *n* steps ahead, `jumpahead(n)` jumps to another state likely to be separated by many steps.

random. `getrandbits(k)`

Returns a python `long` int with *k* random bits. This method is supplied with the MersenneTwister generator and some other generators may also provide it as an optional part of the API. When available, `getrandbits()` enables `randrange()` to handle arbitrarily large ranges.

*New in version 2.4.*

Functions for integers:

random. `randrange(stop)`

random. `randrange(start, stop[, step])`

Return a randomly selected element from `range(start, stop, step)`. This is equivalent to `choice(range(start, stop, step))`, but doesn’t actually build a range object.

*New in version 1.5.2.*

random. `randint(a, b)`
Return a random integer $N$ such that $a \leq N \leq b$.

Functions for sequences:

**random.choice(seq)**

Return a random element from the non-empty sequence `seq`. If `seq` is empty, raises `IndexError`.

**random.shuffle(x[, random])**

Shuffle the sequence `x` in place. The optional argument `random` is a 0-argument function returning a random float in [0.0, 1.0); by default, this is the function `random()`.

Note that for even rather small `len(x)`, the total number of permutations of `x` is larger than the period of most random number generators; this implies that most permutations of a long sequence can never be generated.

**random.sample(population, k)**

Return a `k` length list of unique elements chosen from the population sequence. Used for random sampling without replacement.

*New in version 2.3.*

Returns a new list containing elements from the population while leaving the original population unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples. This allows raffle winners (the sample) to be partitioned into grand prize and second place winners (the subslices).

Members of the population need not be `hashable` or unique. If the population contains repeats, then each occurrence is a possible selection in the sample.
To choose a sample from a range of integers, use an `xrange()` object as an argument. This is especially fast and space efficient for sampling from a large population: `sample(xrange(10000000), 60)`.

The following functions generate specific real-valued distributions. Function parameters are named after the corresponding variables in the distribution’s equation, as used in common mathematical practice; most of these equations can be found in any statistics text.

**random. random()**

Return the next random floating point number in the range [0.0, 1.0).

**random. uniform(a, b)**

Return a random floating point number $N$ such that $a \leq N \leq b$ for $a \leq b$ and $b \leq N \leq a$ for $b < a$.

The end-point value $b$ may or may not be included in the range depending on floating-point rounding in the equation $a + (b-a) \times$ random().

**random. triangular(low, high, mode)**

Return a random floating point number $N$ such that $low \leq N \leq high$ and with the specified mode between those bounds. The low and high bounds default to zero and one. The mode argument defaults to the midpoint between the bounds, giving a symmetric distribution.

*New in version 2.6.*

**random. betavariate(alpha, beta)**

Beta distribution. Conditions on the parameters are $alpha > 0$ and $beta > 0$. Returned values range between 0 and 1.
random. **expovariate**(\(\text{lambd}\))

Exponential distribution. \(\text{lambd}\) is 1.0 divided by the desired mean. It should be nonzero. (The parameter would be called “lambda”, but that is a reserved word in Python.) Returned values range from 0 to positive infinity if \(\text{lambd}\) is positive, and from negative infinity to 0 if \(\text{lambd}\) is negative.

random. **gammavariate**(\(\text{alpha}, \text{beta}\))

Gamma distribution. (Not the gamma function!) Conditions on the parameters are \(\text{alpha} > 0\) and \(\text{beta} > 0\).

The probability distribution function is:

\[
\text{pdf}(x) = \frac{x^{\text{alpha} - 1} \times \text{math.exp(-x / \text{beta})}}{\text{math.gamma(\text{alpha})} \times \text{beta}^{\text{alpha}}}
\]

random. **gauss**(\(\text{mu}, \text{sigma}\))

Gaussian distribution. \(\text{mu}\) is the mean, and \(\text{sigma}\) is the standard deviation. This is slightly faster than the \text{normalvariate()} function defined below.

random. **lognormvariate**(\(\text{mu}, \text{sigma}\))

Log normal distribution. If you take the natural logarithm of this distribution, you'll get a normal distribution with mean \(\text{mu}\) and standard deviation \(\text{sigma}\). \(\text{mu}\) can have any value, and \(\text{sigma}\) must be greater than zero.

random. **normalvariate**(\(\text{mu}, \text{sigma}\))

Normal distribution. \(\text{mu}\) is the mean, and \(\text{sigma}\) is the standard deviation.

random. **vonmisesvariate**(\(\text{mu}, \text{kappa}\))

\(\text{mu}\) is the mean angle, expressed in radians between 0 and 2*\(\pi\), and \(\text{kappa}\) is the concentration parameter, which must be greater than or equal to zero. If \(\text{kappa}\) is equal to zero, this distribution
reduces to a uniform random angle over the range 0 to $2\pi$.

```python
random.paretovariate(alpha)
```

Pareto distribution. *alpha* is the shape parameter.

```python
random.weibullvariate(alpha, beta)
```

Weibull distribution. *alpha* is the scale parameter and *beta* is the shape parameter.

Alternative Generators:

```python
class random.WichmannHill([seed])
```

Class that implements the Wichmann-Hill algorithm as the core generator. Has all of the same methods as `Random` plus the `whseed()` method described below. Because this class is implemented in pure Python, it is not threadsafe and may require locks between calls. The period of the generator is 6,953,607,871,644 which is small enough to require care that two independent random sequences do not overlap.

```python
random.whseed([x])
```

This is obsolete, supplied for bit-level compatibility with versions of Python prior to 2.1. See `seed()` for details. `whseed()` does not guarantee that distinct integer arguments yield distinct internal states, and can yield no more than about $2^{24}$ distinct internal states in all.

```python
class random.SystemRandom([seed])
```

Class that uses the `os.urandom()` function for generating random numbers from sources provided by the operating system. Not available on all systems. Does not rely on software state and sequences are not reproducible. Accordingly, the `seed()` and `jumpahead()` methods have no effect and are ignored. The
getstate() and setstate() methods raise NotImplemented Error if called.

New in version 2.4.

Examples of basic usage:

```python
>>> random.random()  # Random float x, 0.0 <= x < 1.0
0.37444887175646646
>>> random.uniform(1, 10)  # Random float x, 1.0 <= x < 10.0
1.1800146073117523
>>> random.randint(1, 10)  # Integer from 1 to 10, endpoints included
7
>>> random.randrange(0, 101, 2)  # Even integer from 0 to 100
26
>>> random.choice('abcdefghij')  # Choose a random element
'c'

>>> items = [1, 2, 3, 4, 5, 6, 7]
>>> random.shuffle(items)
>>> items
[7, 3, 2, 5, 6, 4, 1]

>>> random.sample([1, 2, 3, 4, 5], 3)  # Choose 3 elements
[4, 1, 5]
```


Complementary-Multiply-with-Carry recipe for a compatible alternative random number generator with a long period and comparatively simple update operations.
This module implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML. Each has been recast in a form suitable for Python.

The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an “iterator algebra” making it possible to construct specialized tools succinctly and efficiently in pure Python.

For instance, SML provides a tabulation tool: \texttt{tabulate(f)} which produces a sequence \( f(0), f(1), \ldots \). The same effect can be achieved in Python by combining \texttt{imap()} and \texttt{count()} to form \texttt{imap(f, count())}.

These tools and their built-in counterparts also work well with the high-speed functions in the \texttt{operator} module. For example, the multiplication operator can be mapped across two vectors to form an efficient dot-product: \texttt{sum(imap(operator.mul, vector1, vector2))}.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Iterator} & \textbf{Arguments} & \textbf{Results} & \textbf{Example} \\
\hline
\texttt{count()} & start, [step] & start, start+step, start+2*step, \ldots & \texttt{count(10)} \rightarrow 10 11 12 13 14 \ldots \\
\hline
\texttt{cycle()} & p & p0, p1, \ldots plast, p0, p1, \ldots & \texttt{cycle('ABCD')} \rightarrow A B C D A B C D \ldots \\
\hline
\texttt{repeat()} & elem [,n] & elem, elem, elem, \ldots endlessly or up to n & \texttt{repeat(10, 3)} \rightarrow \\
\hline
\end{tabular}
\end{table}
Iterators terminating on the shortest input sequence:

<table>
<thead>
<tr>
<th>Iterator</th>
<th>Arguments</th>
<th>Results</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>chain()</td>
<td>p, q, ...</td>
<td>p0, p1, ... plast, q0, q1, ...</td>
<td>chain('ABC', 'DEF') --&gt; A B C D E F</td>
</tr>
<tr>
<td>compress()</td>
<td>data, selectors</td>
<td>(d[0] if s[0]), (d[1] if s[1]), ...</td>
<td>compress('ABCDEF', [1,0,1,0,1,1]) --&gt; C E F</td>
</tr>
<tr>
<td>dropwhile()</td>
<td>pred, seq</td>
<td>seq[n], seq[n+1], starting when pred fails</td>
<td>dropwhile(lambda x: x&lt;5, [1,4,6,4,1]) --&gt; 6 4 1</td>
</tr>
<tr>
<td>groupby()</td>
<td>iterable[, keyfunc]</td>
<td>sub-iterators grouped by value of keyfunc(v)</td>
<td></td>
</tr>
<tr>
<td>ifilter()</td>
<td>pred, seq</td>
<td>elements of seq where pred(elem) is true</td>
<td>ifilter(lambda x: x%2, range(10)) --&gt; 1 3 5 7 9</td>
</tr>
<tr>
<td>ifilterfalse()</td>
<td>pred, seq</td>
<td>elements of seq where pred(elem) is false</td>
<td>ifilterfalse(lambda x: x%2, range(10)) --&gt; 0 2 4 6 8</td>
</tr>
<tr>
<td>islice()</td>
<td>seq, [start,] stop [, step]</td>
<td>elements from seq[start:stop:step]</td>
<td>islice('ABCDEFG', 2, None) --&gt; C D E F G</td>
</tr>
<tr>
<td>imap()</td>
<td>func, p, q, ...</td>
<td>func(p0, q0), func(p1, q1), ...</td>
<td>imap(pow, (2,3,10), (5,2,3)) --&gt; 32 9 1000</td>
</tr>
<tr>
<td>starmap()</td>
<td>func, seq</td>
<td>func(*seq[0]), func(*seq[1]), ...</td>
<td>starmap(pow, [(2,5), (3,2), (10,3)]) --&gt; 32 9 1000</td>
</tr>
<tr>
<td>tee()</td>
<td>it, n</td>
<td>it1, it2, ... itn splits one iterator into n</td>
<td></td>
</tr>
<tr>
<td>takewhile()</td>
<td>pred, seq</td>
<td>seq[0], seq[1], until pred fails</td>
<td>takewhile(lambda x: x&lt;5, [1,4,6,4,1]) --&gt; 1 4</td>
</tr>
<tr>
<td>izip()</td>
<td>p, q, ...</td>
<td>(p[0], q[0]), (p[1], q[1]), ...</td>
<td>izip('ABCD', 'xy') --&gt; Ax By</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>izip_longest('ABCD', 'xy')  --&gt; A B C D E F</td>
</tr>
</tbody>
</table>
### Combinatorial generators:

<table>
<thead>
<tr>
<th>Iterator</th>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>product()</strong></td>
<td>p, q, ... [repeat=1]</td>
<td>cartesian product, equivalent to a nested for-loop</td>
</tr>
<tr>
<td><strong>permutations()</strong></td>
<td>p[, r]</td>
<td>r-length tuples, all possible orderings, no repeated elements</td>
</tr>
<tr>
<td><strong>combinations()</strong></td>
<td>p, r</td>
<td>r-length tuples, in sorted order, no repeated elements</td>
</tr>
<tr>
<td><strong>combinations_with_replacement()</strong></td>
<td>p, r</td>
<td>r-length tuples, in sorted order, with repeated elements</td>
</tr>
</tbody>
</table>

- **product('ABCD', repeat=2)**
  - AA AB AC AD
  - BA BB BC BD
  - CA CB CC CD
  - DA DB DC DD

- **permutations('ABCD', 2)**
  - AB AC AD BA BC BD CA CB CD DA DB DC

- **combinations('ABCD', 2)**
  - AB AC AD BC BD CA CB CD DA DB DC

- **combinations_with_replacement('ABCD', 2)**
  - AA AB AC AD BB BC BD CC
9.7.1. Itertool functions

The following module functions all construct and return iterators. Some provide streams of infinite length, so they should only be accessed by functions or loops that truncate the stream.

**itertools.** *chain(*iterables*)***

Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. Used for treating consecutive sequences as a single sequence. Roughly equivalent to:

```python
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

**classmethod chain.** *from_iterable(iterable)*

Alternate constructor for chain(). Gets chained inputs from a single iterable argument that is evaluated lazily. Roughly equivalent to:

```python
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

*New in version 2.6.*

**itertools.** *combinations(iterable, r)*

Return *r* length subsequences of elements from the input *iterable*.

Combinations are emitted in lexicographic sort order. So, if the input *iterable* is sorted, the combination tuples will be produced in
sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each combination.

Roughly equivalent to:

```python
def combinations(iterable, r):
    # combinations('ABCD', 2) --> AB AC AD BC BD CD
    # combinations(range(4), 3) --> 012 013 023 123
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = range(r)
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

The code for `combinations()` can be also expressed as a subsequence of `permutations()` after filtering entries where the elements are not in sorted order (according to their position in the input pool):

```python
def combinations(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in permutations(range(n), r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

The number of items returned is $n! / r! / (n-r)!$ when $0 \leq r$
<= n or zero when r > n.

*New in version 2.6.*

`itertools.combinations_with_replacement(iterable, r)`

Return r length subsequences of elements from the input *iterable* allowing individual elements to be repeated more than once.

Combinations are emitted in lexicographic sort order. So, if the input *iterable* is sorted, the combination tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, the generated combinations will also be unique.

Roughly equivalent to:

```python
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) --> AA AB AC BB BC CC
    pool = tuple(iterable)
n = len(pool)

    if not n and r:
        return

    indices = [0] * r
    yield tuple(pool[i] for i in indices)

    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return

        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

The code for `combinations_with_replacement()` can be also expressed as a subsequence of `product()` after filtering entries where the elements are not in sorted order (according to their position in the input pool):
def combinations_with_replacement(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in product(range(n), repeat=r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)

The number of items returned is \( \frac{(n+r-1)!}{r! (n-1)!} \) when \( n > 0 \).

*New in version 2.7.*

**itertools.compress(data, selectors)**

Make an iterator that filters elements from *data* returning only those that have a corresponding element in *selectors* that evaluates to `True`. Stops when either the *data* or *selectors* iterables has been exhausted. Roughly equivalent to:

```python
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
    return (d for d, s in izip(data, selectors) if s)
```

*New in version 2.7.*

**itertools.count(start=0, step=1)**

Make an iterator that returns evenly spaced values starting with \( n \). Often used as an argument to `imap()` to generate consecutive data points. Also, used with `izip()` to add sequence numbers. Equivalent to:

```python
def count(start=0, step=1):
    # count(10) --> 10 11 12 13 14 ...
    # count(2.5, 0.5) --> 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

When counting with floating point numbers, better accuracy can
sometimes be achieved by substituting multiplicative code such as: `(start + step * i for i in count())`.

*Changed in version 2.7:* added `step` argument and allowed non-integer arguments.

**itertools.**`cycle(iterable)`

Make an iterator returning elements from the iterable and saving a copy of each. When the iterable is exhausted, return elements from the saved copy. Repeats indefinitely. Roughly equivalent to:

```python
def cycle(iterable):
    # cycle('ABCD') --
    # > A B C D A B C D A B C D ...  
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
            yield element
```

Note, this member of the toolkit may require significant auxiliary storage (depending on the length of the iterable).

**itertools.**`dropwhile(predicate, iterable)`

Make an iterator that drops elements from the iterable as long as the predicate is true; afterwards, returns every element. Note, the iterator does not produce *any* output until the predicate first becomes false, so it may have a lengthy start-up time. Roughly equivalent to:

```python
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --
    # > 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
        break
    for x in iterable:
        yield x
```
The `groupby()` function is an iterator that returns consecutive keys and groups from the `iterable`. The `key` is a function computing a key value for each element. If not specified or is `None`, `key` defaults to an identity function and returns the element unchanged. Generally, the iterable needs to already be sorted on the same key function.

The operation of `groupby()` is similar to the `uniq` filter in Unix. It generates a break or new group every time the value of the key function changes (which is why it is usually necessary to have sorted the data using the same key function). That behavior differs from SQL's GROUP BY which aggregates common elements regardless of their input order.

The returned group is itself an iterator that shares the underlying iterable with `groupby()`. Because the source is shared, when the `groupby()` object is advanced, the previous group is no longer visible. So, if that data is needed later, it should be stored as a list:

```python
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
groups.append(list(g))  # Store group iterator as a
uniquekeys.append(k)
```

`groupby()` is roughly equivalent to:

```python
class groupby(object):
    # [k for k, g in groupby('AAAABBBCCDAABBB')] --> A B C D
    # [list(g) for k, g in groupby('AAAABBBCCD')] --> AAAA BBB B
    def __init__(self, iterable, key=None):
        if key is None:
            key = lambda x: x
        self.keyfunc = key
```
self.it = iter(iterable)
self.tgtkey = self.currkey = self.currvalue = object

def __iter__(self):
    return self
def next(self):
    while self.currkey == self.tgtkey:
        self.currvalue = next(self.it)  # Exit on StopIteration
        self.currkey = self.keyfunc(self.currvalue)
    self.tgtkey = self.currkey
    return (self.currkey, self._grouper(self.tgtkey))

def _grouper(self, tgtkey):
    while self.currkey == tgtkey:
        yield self.currvalue
        self.currvalue = next(self.it)  # Exit on StopIteration
        self.currkey = self.keyfunc(self.currvalue)


New in version 2.4.

`itertools.ifilter(predicate, iterable)`

Make an iterator that filters elements from iterable returning only those for which the predicate is `True`. If `predicate` is `None`, return the items that are true. Roughly equivalent to:

```python
def ifilter(predicate, iterable):
    # ifilter(lambda x: x%2, range(10)) --> 1 3 5 7 9
    if predicate is None:
        predicate = bool
    for x in iterable:
        if predicate(x):
            yield x
```

`itertools.ifilterfalse(predicate, iterable)`

Make an iterator that filters elements from iterable returning only those for which the predicate is `False`. If `predicate` is `None`, return the items that are false. Roughly equivalent to:

```python
def ifilterfalse(predicate, iterable):
    # ifilterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
    if predicate is None:
        predicate = bool
    for x in iterable:
if not predicate(x):
    yield x

**itertools.**\texttt{imap(function, *iterables)}

Make an iterator that computes the function using arguments from each of the iterables. If \texttt{function} is set to \texttt{None}, then \texttt{imap()} returns the arguments as a tuple. Like \texttt{map()} but stops when the shortest iterable is exhausted instead of filling in \texttt{None} for shorter iterables. The reason for the difference is that infinite iterator arguments are typically an error for \texttt{map()} (because the output is fully evaluated) but represent a common and useful way of supplying arguments to \texttt{imap()}.

```python
def imap(function, *iterables):
    # \texttt{imap(pow, (2,3,10), (5,2,3))} \texttt{--> 32 9 1000}
    iterables = map(iter, iterables)
    while True:
        args = [next(it) for it in iterables]
        if function is None:
            yield tuple(args)
        else:
            yield function(*args)
```

**itertools.**\texttt{islice(iterable, stop)}

**itertools.**\texttt{islice(iterable, start, stop[, step])}

Make an iterator that returns selected elements from the iterable. If \texttt{start} is non-zero, then elements from the iterable are skipped until \texttt{start} is reached. Afterward, elements are returned consecutively unless \texttt{step} is set higher than one which results in items being skipped. If \texttt{stop} is \texttt{None}, then iteration continues until the iterator is exhausted, if at all; otherwise, it stops at the specified position. Unlike regular slicing, \texttt{islice()} does not support negative values for \texttt{start}, \texttt{stop}, or \texttt{step}. Can be used to extract related fields from data where the internal structure has been flattened (for example, a multi-line report may list a name field on every third line). Roughly equivalent to:
def islice(iterable, *args):
    # islice('ABCDEFG', 2) --> A B
    # islice('ABCDEFG', 2, 4) --> C D
    # islice('ABCDEFG', 2, None) --> C D E F G
    # islice('ABCDEFG', 0, None, 2) --> A C E G
    s = slice(*args)
    it = iter(xrange(s.start or 0, s.stop or sys.maxint, s.step)
    nexti = next(it)
    for i, element in enumerate(iterable):
        if i == nexti:
            yield element
        nexti = next(it)

If start is None, then iteration starts at zero. If step is None, then the step defaults to one.

Changed in version 2.5: accept None values for default start and step.

itertools.izip(*iterables)
Make an iterator that aggregates elements from each of the iterables. Like zip() except that it returns an iterator instead of a list. Used for lock-step iteration over several iterables at a time. Roughly equivalent to:

def izip(*iterables):
    # izip('ABCD', 'xy') --> Ax By
    iterators = map(iter, iterables)
    while iterators:
        yield tuple(map(next, iterators))

Changed in version 2.4: When no iterables are specified, returns a zero length iterator instead of raising a TypeError exception.

The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using izip(*[iter(s)]*n).
**izip()** should only be used with unequal length inputs when you don’t care about trailing, unmatched values from the longer iterables. If those values are important, use **izip_longest()** instead.

```python
class ZipExhausted(Exception):
    pass

def izip_longest(*args, **kwds):
    # izip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C-
    fillvalue = kwds.get('fillvalue')
    counter = [len(args) - 1]
    def sentinel():
        if not counter[0]:
            raise ZipExhausted
        counter[0] -= 1
        yield fillvalue
    fillers = repeat(fillvalue)
    iterators = [chain(it, sentinel(), fillers) for it in args]
    try:
        while iterators:
            yield tuple(map(next, iterators))
    except ZipExhausted:
        pass
```

If one of the iterables is potentially infinite, then the **izip_longest()** function should be wrapped with something that limits the number of calls (for example **islice()** or **takewhile()**). If not specified, **fillvalue** defaults to **None**.

*New in version 2.6.*
*itertools*. `permutations(iterable[, r])`

Return successive `r` length permutations of elements in the `iterable`.

If `r` is not specified or is `None`, then `r` defaults to the length of the `iterable` and all possible full-length permutations are generated.

Permutations are emitted in lexicographic sort order. So, if the input `iterable` is sorted, the permutation tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each permutation.

Roughly equivalent to:

```python
def permutations(iterable, r=None):
    # permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD
    # permutations(range(3)) --> 012 021 102 120 201 210
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return
    indices = range(n)
    cycles = range(n, n-r, -1)
    yield tuple(pool[i] for i in indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

The code for `permutations()` can be also expressed as a subsequence of `product()`, filtered to exclude entries with repeated elements (those from the same position in the input pool):

```python
def permutations(iterable, r=None):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    for indices in product(range(n), repeat=r):
        if len(set(indices)) == r:
            yield tuple(pool[i] for i in indices)
```

The number of items returned is \( n! / (n-r)! \) when \( 0 \leq r \leq n \) or zero when \( r > n \).

*New in version 2.6.*

`itertools.product(*iterables[, repeat])`

Cartesian product of input iterables.

Roughly equivalent to nested for-loops in a generator expression. For example, `product(A, B)` returns the same as `((x, y) for x in A for y in B)`.

The nested loops cycle like an odometer with the rightmost element advancing on every iteration. This pattern creates a lexicographic ordering so that if the input's iterables are sorted, the product tuples are emitted in sorted order.

To compute the product of an iterable with itself, specify the number of repetitions with the optional `repeat` keyword argument. For example, `product(A, repeat=4)` means the same as `product(A, A, A, A)`. 
This function is roughly equivalent to the following code, except that the actual implementation does not build up intermediate results in memory:

```python
def product(*args, **kwds):
    # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) --> 000 001 010 011 100 101
    pools = map(tuple, args) * kwds.get('repeat', 1)
    result = [[]]
    for pool in pools:
        result = [x+y for x in result for y in pool]
    for prod in result:
        yield tuple(prod)
```

*New in version 2.6.*

`itertools.repeat(object[, times])`

Make an iterator that returns `object` over and over again. Runs indefinitely unless the `times` argument is specified. Used as argument to `imap()` for invariant function parameters. Also used with `izip()` to create constant fields in a tuple record. Roughly equivalent to:

```python
def repeat(object, times=None):
    # repeat(10, 3) --> 10 10 10
    if times is None:
        while True:
            yield object
    else:
        for i in xrange(times):
            yield object
```

A common use for `repeat` is to supply a stream of constant values to `imap` or `zip`:

```python
>>> list(imap(pow, xrange(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`itertools.starmap(function, iterable)`
Make an iterator that computes the function using arguments obtained from the iterable. Used instead of `imap()` when argument parameters are already grouped in tuples from a single iterable (the data has been “pre-zipped”). The difference between `imap()` and `starmap()` parallels the distinction between `function(a, b)` and `function(*c)`. Roughly equivalent to:

```python
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

*Changed in version 2.6:* Previously, `starmap()` required the function arguments to be tuples. Now, any iterable is allowed.

**itertools.** `takewhile(predicate, iterable)`

Make an iterator that returns elements from the iterable as long as the predicate is true. Roughly equivalent to:

```python
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

**itertools.** `tee(iterable[, n=2])`

Return `n` independent iterators from a single iterable. Roughly equivalent to:

```python
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:
                # when the local deque is empty
                newval = next(it)
                # fetch a new value
            for d in deques:
                # load it to all the deques
```
```python
d.append(newval)
    yield mydeque.popleft()
return tuple(gen(d) for d in deques)
```

Once `tee()` has made a split, the original `iterable` should not be used anywhere else; otherwise, the `iterable` could get advanced without the tee objects being informed.

This itertool may require significant auxiliary storage (depending on how much temporary data needs to be stored). In general, if one iterator uses most or all of the data before another iterator starts, it is faster to use `list()` instead of `tee()`.

*New in version 2.4.*
This section shows recipes for creating an extended toolset using the existing itertools as building blocks.

The extended tools offer the same high performance as the underlying toolset. The superior memory performance is kept by processing elements one at a time rather than bringing the whole iterable into memory all at once. Code volume is kept small by linking the tools together in a functional style which helps eliminate temporary variables. High speed is retained by preferring “vectorized” building blocks over the use of for-loops and generators which incur interpreter overhead.

```python
def take(n, iterable):
    "Return first n items of the iterable as a list"
    return list(islice(iterable, n))

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return imap(function, count(start))

def consume(iterator, n):
    "Advance the iterator n-steps ahead. If n is none, consume # Use functions that consume iterators at C speed.
    if n is None:  
        # feed the entire iterator into a zero-length deque
collections.deque(iterator, maxlen=0)
    else:
        # advance to the empty slice starting at position n
    next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
    "Returns the nth item or a default value"
    return next(islice(iterable, n, None), default)

def all_equal(iterable):
    "Returns True if all the elements are equal to each other"
g = groupby(iterable)
    return next(g, True) and not next(g, False)
```
def quantify(iterable, pred=bool):
    "Count how many times the predicate is true"
    return sum(imap(pred, iterable))

def padnone(iterable):
    """Returns the sequence elements and then returns None indefinitely.
    Useful for emulating the behavior of the built-in map() function.
    """
    return chain(iterable, repeat(None))

def ncycles(iterable, n):
    "Returns the sequence elements n times"
    return chain.from_iterable(repeat(tuple(iterable), n))

def dotproduct(vec1, vec2):
    return sum(imap(operator.mul, vec1, vec2))

def flatten(listOfLists):
    "Flatten one level of nesting"
    return chain.from_iterable(listOfLists)

def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments."
    Example: repeatfunc(random.random)
    ""
    if times is None:
        return starmap(func, repeat(args))
    return starmap(func, repeat(args, times))

def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b, None)
    return izip(a, b)

def grouper(iterable, n, fillvalue=None):
    "Collect data into fixed-length chunks or blocks"
    # grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx
    args = [iter(iterable)] * n
    return izip_longest(fillvalue=fillvalue, *args)

def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    # Recipe credited to George Sakkis
pending = len(iterables)
nexts = cycle(iter(it).next for it in iterables)
while pending:
    try:
        for next in nexts:
            yield next()
    except StopIteration:
        pending -= 1
        nexts = cycle(islice(nexts, pending))

def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3)
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)))

def unique_everseen(iterable, key=None):
    "List unique elements, preserving order. Remember all eleme
    # unique_everseen('AAAAABBCCDAA BBB') --> A B C D
    # unique_everseen('ABB CcAD', str.lower) --> A B C D
    seen = set()
    seen_add = seen.add
    if key is None:
        for element in ifilterfalse(seen.__contains__, iterable):
            seen_add(element)
            yield element
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen_add(k)
                yield element

def unique_justseen(iterable, key=None):
    "List unique elements, preserving order. Remember only the
    # unique_justseen('AAAA BBB C C D A A B B B') --> A B C D A B
    # unique_justseen('ABB CcAD', str.lower) --> A B C A D
    return imap(next, imap(itemgetter(1), groupby(iterable, key)))

def iter_except(func, exception, first=None):
    """ Call a function repeatedly until an exception is raised

    Converts a call-until-exception interface to an iterator in
    Like __builtin__.iter(func, sentinel) but uses an exception of
    a sentinel to end the loop.

    Examples:
    bsddbiter = iter_except(db.next, bsddb.error, db.first)""
    try:
        yield from func() if first is None
        for result in func():
            yield result
    except exception:
        pass
heapiter = iter_except(functools.partial(heappop, h), IndexError)
dictiter = iter_except(d.popitem, KeyError)
dequeueiter = iter_except(d.popleft, IndexError)
queueiter = iter_except(q.get_nowait, Queue.Empty)
setiter = iter_except(s.pop, KeyError)

""
try:
    if first is not None:
        yield first()
    while 1:
        yield func()
except exception:
    pass

def random_product(*args, **kwds):
    "Random selection from itertools.product(*args, **kwds)"
    pools = map(tuple, args) * kwds.get('repeat', 1)
    return tuple(random.choice(pool) for pool in pools)

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(xrange(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with Replacement(iterable, r):
    "Random selection from itertools.combinations_with_replacement"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.randrange(n) for i in xrange(r))
    return tuple(pool[i] for i in indices)

def tee_lookahead(t, i):
    """Inspect the i-th upcoming value from a tee object while leaving the tee object at its current position."
    Raise an IndexError if the underlying iterator doesn't have enough values."""
for value in islice(t.__copy__(), i, None):
    return value
raise IndexError(i)

Note, many of the above recipes can be optimized by replacing global lookups with local variables defined as default values. For example, the *dotproduct* recipe can be written as:

def dotproduct(vec1, vec2, sum=sum, imap=imap, mul=operator.mul):
    return sum(imap(mul, vec1, vec2))
9.8. **functools — Higher-order functions and operations on callable objects**

*New in version 2.5.*

**Source code:** Lib/functools.py

The *functools* module is for higher-order functions: functions that act on or return other functions. In general, any callable object can be treated as a function for the purposes of this module.

The *functools* module defines the following functions:

**functools.**`cmp_to_key(func)`

Transform an old-style comparison function to a *key function*. Used with tools that accept key functions (such as `sorted()`, `min()`, `max()`, `heapq.nlargest()`, `heapq.nsmallest()`, `itertools.groupby()`). This function is primarily used as a transition tool for programs being converted to Python 3 where comparison functions are no longer supported.

A comparison function is any callable that accept two arguments, compares them, and returns a negative number for less-than, zero for equality, or a positive number for greater-than. A key function is a callable that accepts one argument and returns another value to be used as the sort key.

Example:

```python
sorted(iterable, key=cmp_to_key(locale.strcoll))  # locale-a
```
For sorting examples and a brief sorting tutorial, see Sorting HOW TO.

New in version 2.7.

```
functools.total_ordering(cls)
```

Given a class defining one or more rich comparison ordering methods, this class decorator supplies the rest. This simplifies the effort involved in specifying all of the possible rich comparison operations:

The class must define one of __lt__(), __le__(), __gt__(), or __ge__(). In addition, the class should supply an __eq__() method.

For example:

```python
@total_ordering
class Student:
    def __eq__(self, other):
        return ((self.lastname.lower(), self.firstname.lower(),
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        return ((self.lastname.lower(), self.firstname.lower(),
                (other.lastname.lower(), other.firstname.lower(),
```

New in version 2.7.

```
functools.reduce(function, iterable[, initializer])
```

This is the same function as reduce(). It is made available in this module to allow writing code more forward-compatible with Python 3.

New in version 2.6.

```
functools.partial(func[*args][, **keywords])
```

Return a new partial object which when called will behave like
func called with the positional arguments args and keyword arguments keywords. If more arguments are supplied to the call, they are appended to args. If additional keyword arguments are supplied, they extend and override keywords. Roughly equivalent to:

```python
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*(args + fargs), **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

The partial() is used for partial function application which “freezes” some portion of a function’s arguments and/or keywords resulting in a new object with a simplified signature. For example, partial() can be used to create a callable that behaves like the int() function where the base argument defaults to two:

```python
>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

functools.update_wrapper(wrapper, wrapped[, assigned][, updated])

Update a wrapper function to look like the wrapped function. The optional arguments are tuples to specify which attributes of the original function are assigned directly to the matching attributes on the wrapper function and which attributes of the wrapper function are updated with the corresponding attributes from the original function. The default values for these arguments are the module level constants Wilkinson ASSIGNMENTS (which
assigns to the wrapper function’s __name__, __module__ and __doc__, the documentation string) and WRAPPER_UPDATES (which updates the wrapper function’s __dict__, i.e. the instance dictionary).

The main intended use for this function is in decorator functions which wrap the decorated function and return the wrapper. If the wrapper function is not updated, the metadata of the returned function will reflect the wrapper definition rather than the original function definition, which is typically less than helpful.

functools.**wraps**(*wrapped[, assigned[, updated]])

This is a convenience function for invoking update_wrapper() as a function decorator when defining a wrapper function. It is equivalent to partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated). For example:

```python
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print 'Calling decorated function'
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     '''Docstring'''
...     print 'Called example function'
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

Without the use of this decorator factory, the name of the example function would have been 'wrapper', and the docstring
of the original `example()` would have been lost.
9.8.1. partial Objects

**partial** objects are callable objects created by `partial()`. They have three read-only attributes:

**partial.func**
A callable object or function. Calls to the **partial** object will be forwarded to **func** with new arguments and keywords.

**partial.args**
The leftmost positional arguments that will be prepended to the positional arguments provided to a **partial** object call.

**partial.keywords**
The keyword arguments that will be supplied when the **partial** object is called.

**partial** objects are like **function** objects in that they are callable, weak referencable, and can have attributes. There are some important differences. For instance, the `__name__` and `__doc__` attributes are not created automatically. Also, **partial** objects defined in classes behave like static methods and do not transform into bound methods during instance attribute look-up.
9.9. operator — Standard operators as functions

The `operator` module exports a set of efficient functions corresponding to the intrinsic operators of Python. For example, `operator.add(x, y)` is equivalent to the expression `x+y`. The function names are those used for special class methods; variants without leading and trailing `__` are also provided for convenience.

The functions fall into categories that perform object comparisons, logical operations, mathematical operations, sequence operations, and abstract type tests.

The object comparison functions are useful for all objects, and are named after the rich comparison operators they support:

```python
operator.lt(a, b)
operator.le(a, b)
operator.eq(a, b)
operator.ne(a, b)
operator.ge(a, b)
operator.gt(a, b)
operator.__lt__(a, b)
operator.__le__(a, b)
operator.__eq__(a, b)
operator.__ne__(a, b)
operator.__ge__(a, b)
operator.__gt__(a, b)
```

Perform “rich comparisons” between `a` and `b`. Specifically, `lt(a, b)` is equivalent to `a < b`, `le(a, b)` is equivalent to `a <= b`, `eq(a, b)` is equivalent to `a == b`, `ne(a, b)` is equivalent to `a != b`, `gt(a, b)` is equivalent to `a > b` and `ge(a, b)` is equivalent to `a >= b`. Note that unlike the built-in `cmp()`, these functions can return any
value, which may or may not be interpretable as a Boolean value. See \texttt{Comparisons} for more information about rich comparisons.

\textit{New in version 2.2.}

The logical operations are also generally applicable to all objects, and support truth tests, identity tests, and boolean operations:

\begin{verbatim}
operator.\texttt{not\_}(obj)
operator._\texttt{not\_}(obj)
\end{verbatim}

Return the outcome of \texttt{not obj}. (Note that there is no \texttt{\_\_not\_\_()} method for object instances; only the interpreter core defines this operation. The result is affected by the \texttt{\_\_nonzero\_\_()} and \texttt{\_\_len\_\_()} methods.)

\begin{verbatim}
operator.\texttt{truth\_}(obj)
\end{verbatim}

Return \texttt{True} if \texttt{obj} is true, and \texttt{False} otherwise. This is equivalent to using the \texttt{bool} constructor.

\begin{verbatim}
operator.\texttt{is\_}(a, b)
\end{verbatim}

Return \texttt{a is b}. Tests object identity.

\textit{New in version 2.3.}

\begin{verbatim}
operator.\texttt{is\_not\_}(a, b)
\end{verbatim}

Return \texttt{a is not b}. Tests object identity.

\textit{New in version 2.3.}

The mathematical and bitwise operations are the most numerous:

\begin{verbatim}
operator.\texttt{abs\_}(obj)
operator._\texttt{abs\_}(obj)
\end{verbatim}

Return the absolute value of \texttt{obj}.

\begin{verbatim}
operator.\texttt{add\_}(a, b)
\end{verbatim}
operator.__add__(a, b)
    Return \(a + b\), for \(a\) and \(b\) numbers.

operator.and_(a, b)
operator.__and__(a, b)
    Return the bitwise and of \(a\) and \(b\).

operator.div(a, b)
operator.__div__(a, b)
    Return \(a / b\) when \(__future__.division\) is not in effect. This is also known as “classic” division.

operator.floordiv(a, b)
operator.__floordiv__(a, b)
    Return \(a // b\).

New in version 2.2.

operator.index(a)
operator.__index__(a)
    Return \(a\) converted to an integer. Equivalent to \(a.__index__()\).

New in version 2.5.

operator.inv(obj)
operator.invert(obj)
operator.__inv__(obj)
operator.__invert__(obj)
    Return the bitwise inverse of the number \(obj\). This is equivalent to \(
\sim obj\).

New in version 2.0: The names \(invert()\) and \(__invert__()\).

operator.lshift(a, b)
operator.__lshift__(a, b)
    Return \(a\) shifted left by \(b\).
operator.mod(a, b)  
operator.__mod__(a, b)  
    Return a % b.

operator.mul(a, b)  
operator.__mul__(a, b)  
    Return a * b, for a and b numbers.

operator.neg(obj)  
operator.__neg__(obj)  
    Return obj negated (-obj).

operator.or_(a, b)  
operator.__or__(a, b)  
    Return the bitwise or of a and b.

operator.pos(obj)  
operator.__pos__(obj)  
    Return obj positive (+obj).

operator.pow(a, b)  
operator.__pow__(a, b)  
    Return a ** b, for a and b numbers.

    New in version 2.3.

operator.rshift(a, b)  
operator.__rshift__(a, b)  
    Return a shifted right by b.

operator.sub(a, b)  
operator.__sub__(a, b)  
    Return a - b.

operator.truediv(a, b)  
operator.__truediv__(a, b)
Return \( a / b \) when `__future__.division` is in effect. This is also known as "true" division.

*New in version 2.2.*

```python
operator.xor(a, b)
operator.__xor__(a, b)
```

Return the bitwise exclusive or of \( a \) and \( b \).

Operations which work with sequences (some of them with mappings too) include:

```python
operator.concat(a, b)
operator.__concat__(a, b)
```

Return \( a + b \) for \( a \) and \( b \) sequences.

```python
operator.contains(a, b)
operator.__contains__(a, b)
```

Return the outcome of the test \( b \ in \ a \). Note the reversed operands.

*New in version 2.0: The name `__contains__()`.*

```python
operator.countOf(a, b)
```

Return the number of occurrences of \( b \) in \( a \).

```python
operator.delitem(a, b)
operator.__delitem__(a, b)
```

Remove the value of \( a \) at index \( b \).

```python
operator.delslice(a, b, c)
operator.__delslice__(a, b, c)
```

Delete the slice of \( a \) from index \( b \) to index \( c-1 \).

*Deprecated since version 2.6: This function is removed in Python 3.x. Use `delitem()` with a slice index.*
operator.getitem(a, b)
operator.__getitem__(a, b)
    Return the value of a at index b.

operator.getslice(a, b, c)
operator.__getslice__(a, b, c)
    Return the slice of a from index b to index c-1.

<TextView>
Deprecation since version 2.6: This function is removed in Python 3.x. Use getitem() with a slice index.
</TextView>

operator.indexOf(a, b)
operator.indexOf(a, b)
    Return the index of the first of occurrence of b in a.

operator.repeat(a, b)
operator.__repeat__(a, b)
    Deprecated since version 2.7: Use __mul__() instead.
    Return a * b where a is a sequence and b is an integer.

operator.sequenceIncludes(...)
    Deprecated since version 2.0: Use contains() instead.
    Alias for contains().

operator.setitem(a, b, c)
operator.__setitem__(a, b, c)
    Set the value of a at index b to c.

operator.setslice(a, b, c, v)
operator.__setslice__(a, b, c, v)
    Set the slice of a from index b to index c-1 to the sequence v.

<TextView>
Deprecation since version 2.6: This function is removed in
</TextView>
Python 3.x. Use `setitem()` with a slice index.

Example use of operator functions:

```python
>>> # Elementwise multiplication
>>> map(mul, [0, 1, 2, 3], [10, 20, 30, 40])
[0, 20, 60, 120]
>>> # Dot product
>>> sum(map(mul, [0, 1, 2, 3], [10, 20, 30, 40]))
200
```

Many operations have an “in-place” version. The following functions provide a more primitive access to in-place operators than the usual syntax does; for example, the statement `x += y` is equivalent to `x = operator.iadd(x, y)`. Another way to put it is to say that `z = operator.iadd(x, y)` is equivalent to the compound statement `z = x; z += y`.

```python
operator.iadd(a, b)
operator.__iadd__(a, b)
    a = iadd(a, b) is equivalent to a += b.

New in version 2.5.
```

```python
operator.iand(a, b)
operator.__iand__(a, b)
    a = iand(a, b) is equivalent to a &= b.

New in version 2.5.
```

```python
operator.iconcat(a, b)
operator.__iconcat__(a, b)
    a = iconcat(a, b) is equivalent to a += b for a and b sequences.

New in version 2.5.
```
operator.\texttt{idiv}(a, b)
operator.\texttt{\_\_idiv\_}(a, b)

\begin{verbatim}
a = idiv(a, b) is equivalent to a /= b when \texttt{\_\_future\_.division} is not in effect.
\end{verbatim}

\textit{New in version 2.5.}

operator.\texttt{ifloordiv}(a, b)
operator.\texttt{\_\_ifloordiv\_}(a, b)

\begin{verbatim}
a = ifloordiv(a, b) is equivalent to a //= b.
\end{verbatim}

\textit{New in version 2.5.}

operator.\texttt{ilshift}(a, b)
operator.\texttt{\_\_ilshift\_}(a, b)

\begin{verbatim}
a = ilshift(a, b) is equivalent to a <<= b.
\end{verbatim}

\textit{New in version 2.5.}

operator.\texttt{imod}(a, b)
operator.\texttt{\_\_imod\_}(a, b)

\begin{verbatim}
a = imod(a, b) is equivalent to a %= b.
\end{verbatim}

\textit{New in version 2.5.}

operator.\texttt{imul}(a, b)
operator.\texttt{\_\_imul\_}(a, b)

\begin{verbatim}
a = imul(a, b) is equivalent to a *= b.
\end{verbatim}

\textit{New in version 2.5.}

operator.\texttt{ior}(a, b)
operator.\texttt{\_\_ior\_}(a, b)

\begin{verbatim}
a = ior(a, b) is equivalent to a |= b.
\end{verbatim}

\textit{New in version 2.5.}
operator.\texttt{ipow}(a, b)\hfill \texttt{operator.\_ipow\_}(a, b)\hfill \texttt{a} = \texttt{ipow}(a, b) \text{ is equivalent to } a \texttt{**= b}.\hfill \\

\textit{New in version 2.5.}\hfill \\

operator.\texttt{irepeat}(a, b)\hfill \texttt{operator.\_irepeat\_}(a, b)\hfill \texttt{a} = \texttt{irepeat}(a, b) \text{ is equivalent to } a \texttt{*= b} \text{ where } a \text{ is a sequence and } b \text{ is an integer}.\hfill \\

\textit{Deprecated since version 2.7: Use \texttt{\_imul\_()} instead.}\hfill \\

\textit{New in version 2.5.}\hfill \\

operator.\texttt{irshift}(a, b)\hfill \texttt{operator.\_irshift\_}(a, b)\hfill \texttt{a} = \texttt{irshift}(a, b) \text{ is equivalent to } a \texttt{\text{>= b}}.\hfill \\

\textit{New in version 2.5.}\hfill \\

operator.\texttt{isub}(a, b)\hfill \texttt{operator.\_isub\_}(a, b)\hfill \texttt{a} = \texttt{isub}(a, b) \text{ is equivalent to } a \texttt{-= b}.\hfill \\

\textit{New in version 2.5.}\hfill \\

operator.\texttt{itruediv}(a, b)\hfill \texttt{operator.\_itruediv\_}(a, b)\hfill \texttt{a} = \texttt{itruediv}(a, b) \text{ is equivalent to } a \texttt{/= b} \text{ when } \texttt{\_future\_.division} \text{ is in effect}.\hfill \\

\textit{New in version 2.5.}\hfill \\

operator.\texttt{ixor}(a, b)\hfill \texttt{operator.\_ixor\_}(a, b)\hfill \\

a = ixor(a, b) is equivalent to a ^= b.

New in version 2.5.

The `operator` module also defines a few predicates to test the type of objects; however, these are not all reliable. It is preferable to test abstract base classes instead (see `collections` and `numbers` for details).

**operator.**`isCallable(obj)`

*Deprecated since version 2.0:* Use `isinstance(x, collections.Callable)` instead.

Returns true if the object `obj` can be called like a function, otherwise it returns false. True is returned for functions, bound and unbound methods, class objects, and instance objects which support the `__call__()` method.

**operator.**`isMappingType(obj)`

*Deprecated since version 2.7:* Use `isinstance(x, collections.Mapping)` instead.

Returns true if the object `obj` supports the mapping interface. This is true for dictionaries and all instance objects defining `__getitem__()`.

**operator.**`isNumberType(obj)`

*Deprecated since version 2.7:* Use `isinstance(x, numbers.Number)` instead.

Returns true if the object `obj` represents a number. This is true for all numeric types implemented in C.

**operator.**`isSequenceType(obj)`

*Deprecated since version 2.7:* Use `isinstance(x, collections.Sequence)` instead.

Returns true if the object `obj` is a sequence type.
Returns true if the object obj supports the sequence protocol. This returns true for all objects which define sequence methods in C, and for all instance objects defining __getitem__().

The operator module also defines tools for generalized attribute and item lookups. These are useful for making fast field extractors as arguments for map(), sorted(), itertools.groupby(), or other functions that expect a function argument.

operator.attrgetter(attr)
operator.attrgetter(*attrs)

Return a callable object that fetches attr from its operand. If more than one attribute is requested, returns a tuple of attributes. The attribute names can also contain dots. For example:

- After f = attrgetter('name'), the call f(b) returns b.name.
- After f = attrgetter('name', 'date'), the call f(b) returns (b.name, b.date).
- After f = attrgetter('name.first', 'name.last'), the call f(b) returns (b.name.first, b.name.last).

Equivalent to:

```python
def attrgetter(*items):
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g

def resolve_attr(obj, attr):
    for name in attr.split('.'):  # Split attribute names by dot
        obj = getattr(obj, name)
    return obj
```
New in version 2.4.

Changed in version 2.5: Added support for multiple attributes.

Changed in version 2.6: Added support for dotted attributes.

```
operator.itemgetter(item)
operator.itemgetter(*items)
```

Return a callable object that fetches `item` from its operand using
the operand’s `__getitem__()` method. If multiple items are
specified, returns a tuple of lookup values. For example:

- After `f = itemgetter(2)`, the call `f(r)` returns `r[2]`.
- After `g = itemgetter(2, 5, 3)`, the call `g(r)` returns `(r[2],
r[5], r[3])`.

Equivalent to:

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

The items can be any type accepted by the operand’s
`__getitem__()` method. Dictionaries accept any hashable value.
Lists, tuples, and strings accept an index or a slice:

```
>>> itemgetter(1)(['ABCDEFG'])
'B'
>>> itemgetter(1,3,5)(['ABCDEFG'])
('B', 'D', 'F')
>>> itemgetter(slice(2,None))('ABCDEFG')
'CDEFG'
```
New in version 2.4.

Changed in version 2.5: Added support for multiple item extraction.

Example of using `itemgetter()` to retrieve specific fields from a tuple record:

```python
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> map(getcount, inventory)
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

operator. `methodcaller(name[, args...])`

Return a callable object that calls the method `name` on its operand. If additional arguments and/or keyword arguments are given, they will be given to the method as well. For example:

- After `f = methodcaller('name')`, the call `f(b)` returns `b.name()`.
- After `f = methodcaller('name', 'foo', bar=1)`, the call `f(b)` returns `b.name('foo', bar=1).

Equivalent to:

```python
def methodcaller(name, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

New in version 2.6.
9.9.1. Mapping Operators to Functions

This table shows how abstract operations correspond to operator symbols in the Python syntax and the functions in the `operator` module.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td><code>a + b</code></td>
<td><code>add(a, b)</code></td>
</tr>
<tr>
<td>Concatenation</td>
<td><code>seq1 + seq2</code></td>
<td><code>concat(seq1, seq2)</code></td>
</tr>
<tr>
<td>Containment Test</td>
<td><code>obj in seq</code></td>
<td><code>contains(seq, obj)</code></td>
</tr>
<tr>
<td>Division</td>
<td><code>a / b</code></td>
<td><code>div(a, b)</code> (without <code>__future__.division</code>)</td>
</tr>
<tr>
<td>Division</td>
<td><code>a / b</code></td>
<td><code>truediv(a, b)</code> (with <code>__future__.division</code>)</td>
</tr>
<tr>
<td>Division</td>
<td><code>a // b</code></td>
<td><code>floordiv(a, b)</code></td>
</tr>
<tr>
<td>Bitwise And</td>
<td><code>a &amp; b</code></td>
<td><code>and_(a, b)</code></td>
</tr>
<tr>
<td>Bitwise Exclusive Or</td>
<td><code>a ^ b</code></td>
<td><code>xor(a, b)</code></td>
</tr>
<tr>
<td>Bitwise Inversion</td>
<td><code>~ a</code></td>
<td><code>invert(a)</code></td>
</tr>
<tr>
<td>Bitwise Or</td>
<td>`a</td>
<td>b`</td>
</tr>
<tr>
<td>Exponentiation</td>
<td><code>a ** b</code></td>
<td><code>pow(a, b)</code></td>
</tr>
<tr>
<td>Identity</td>
<td><code>a is b</code></td>
<td><code>is_(a, b)</code></td>
</tr>
<tr>
<td>Identity</td>
<td><code>a is not b</code></td>
<td><code>is_not(a, b)</code></td>
</tr>
<tr>
<td>Indexed Assignment</td>
<td><code>obj[k] = v</code></td>
<td><code>setitem(obj, k, v)</code></td>
</tr>
<tr>
<td>Indexed Deletion</td>
<td><code>del obj[k]</code></td>
<td><code>delitem(obj, k)</code></td>
</tr>
<tr>
<td>Indexing</td>
<td><code>obj[k]</code></td>
<td><code>getitem(obj, k)</code></td>
</tr>
<tr>
<td>Left Shift</td>
<td><code>a &lt;&lt; b</code></td>
<td><code>lshift(a, b)</code></td>
</tr>
<tr>
<td>Modulo</td>
<td><code>a % b</code></td>
<td><code>mod(a, b)</code></td>
</tr>
<tr>
<td>Operation</td>
<td>Expression</td>
<td>Python Function</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Multiplication</td>
<td>(a \times b)</td>
<td><code>mul(a, b)</code></td>
</tr>
<tr>
<td>Negation (Arithmetic)</td>
<td>(-a)</td>
<td><code>neg(a)</code></td>
</tr>
<tr>
<td>Negation (Logical)</td>
<td><code>not a</code></td>
<td><code>not_(a)</code></td>
</tr>
<tr>
<td>Positive</td>
<td>(+ a)</td>
<td><code>pos(a)</code></td>
</tr>
<tr>
<td>Right Shift</td>
<td>(a &gt;&gt; b)</td>
<td><code>rshift(a, b)</code></td>
</tr>
<tr>
<td>Sequence Repetition</td>
<td><code>seq * i</code></td>
<td><code>repeat(seq, i)</code></td>
</tr>
<tr>
<td>Slice Assignment</td>
<td><code>seq[i:j] = values</code></td>
<td><code>setitem(seq, slice(i, j), values)</code></td>
</tr>
<tr>
<td>Slice Deletion</td>
<td><code>del seq[i:j]</code></td>
<td><code>delitem(seq, slice(i, j))</code></td>
</tr>
<tr>
<td>Slicing</td>
<td><code>seq[i:j]</code></td>
<td><code>getitem(seq, slice(i, j))</code></td>
</tr>
<tr>
<td>String Formatting</td>
<td><code>s % obj</code></td>
<td><code>mod(s, obj)</code></td>
</tr>
<tr>
<td>Subtraction</td>
<td>(a - b)</td>
<td><code>sub(a, b)</code></td>
</tr>
<tr>
<td>Truth Test</td>
<td><code>obj</code></td>
<td><code>truth(obj)</code></td>
</tr>
<tr>
<td>Ordering</td>
<td>(a &lt; b)</td>
<td><code>lt(a, b)</code></td>
</tr>
<tr>
<td>Ordering</td>
<td>(a \leq b)</td>
<td><code>le(a, b)</code></td>
</tr>
<tr>
<td>Equality</td>
<td>(a == b)</td>
<td><code>eq(a, b)</code></td>
</tr>
<tr>
<td>Difference</td>
<td>(a \neq b)</td>
<td><code>ne(a, b)</code></td>
</tr>
<tr>
<td>Ordering</td>
<td>(a &gt; b)</td>
<td><code>gt(a, b)</code></td>
</tr>
<tr>
<td>Ordering</td>
<td>(a \geq b)</td>
<td><code>ge(a, b)</code></td>
</tr>
</tbody>
</table>
10. File and Directory Access

The modules described in this chapter deal with disk files and directories. For example, there are modules for reading the properties of files, manipulating paths in a portable way, and creating temporary files. The full list of modules in this chapter is:

- **10.1. os.path** — Common pathname manipulations
- **10.2. fileinput** — Iterate over lines from multiple input streams
- **10.3. stat** — Interpreting stat() results
- **10.4. statvfs** — Constants used with os.statvfs()
- **10.5. filecmp** — File and Directory Comparisons
  - **10.5.1. The dircmp class**
- **10.6. tempfile** — Generate temporary files and directories
- **10.7. glob** — Unix style pathname pattern expansion
- **10.8. fnmatch** — Unix filename pattern matching
- **10.9. linecache** — Random access to text lines
- **10.10. shutil** — High-level file operations
  - **10.10.1. Directory and files operations**
    - **10.10.1.1. copytree example**
  - **10.10.2. Archiving operations**
    - **10.10.2.1. Archiving example**
- **10.11. dircache** — Cached directory listings
- **10.12. macpath** — Mac OS 9 path manipulation functions

See also:

**Section File Objects**
A description of Python's built-in file objects.

**Module os**
Operating system interfaces, including functions to work with files at a lower level than the built-in file object.
10.1. os.path — Common pathname manipulations

This module implements some useful functions on pathnames. To read or write files see open(), and for accessing the filesystem see the os module.

**Note:** On Windows, many of these functions do not properly support UNC pathnames. splitunc() and ismount() do handle them correctly.

Unlike a unix shell, Python does not do any automatic path expansions. Functions such as expanduser() and expandvars() can be invoked explicitly when an application desires shell-like path expansion. (See also the glob module.)

**Note:** Since different operating systems have different path name conventions, there are several versions of this module in the standard library. The os.path module is always the path module suitable for the operating system Python is running on, and therefore usable for local paths. However, you can also import and use the individual modules if you want to manipulate a path that is always in one of the different formats. They all have the same interface:

- posixpath for UNIX-style paths
- ntpath for Windows paths
- macpath for old-style MacOS paths
- os2emxpath for OS/2 EMX paths

os.path.abspath(path)
Return a normalized absolutized version of the pathname `path`. On most platforms, this is equivalent to calling the function `normpath()` as follows: `normpath(join(os.getcwd(), path))`.

*New in version 1.5.2.*

**os.path.basename(path)**

Return the base name of pathname `path`. This is the second element of the pair returned by passing `path` to the function `split()`. Note that the result of this function is different from the Unix `basename` program; where `basename` for `/foo/bar/` returns `'bar'`, the `basename()` function returns an empty string (`''`).

**os.path.commonprefix(list)**

Return the longest path prefix (taken character-by-character) that is a prefix of all paths in `list`. If `list` is empty, return the empty string (`''`). Note that this may return invalid paths because it works a character at a time.

**os.path.dirname(path)**

Return the directory name of pathname `path`. This is the first element of the pair returned by passing `path` to the function `split()`.

**os.path.exists(path)**

Return `True` if `path` refers to an existing path. Returns `False` for broken symbolic links. On some platforms, this function may return `False` if permission is not granted to execute `os.stat()` on the requested file, even if the `path` physically exists.

**os.path.lexists(path)**

Return `True` if `path` refers to an existing path. Returns `True` for broken symbolic links. Equivalent to `exists()` on platforms
lacking `os.lstat()`.

*New in version 2.4.*

**os.path.expanduser(path)**

On Unix and Windows, return the argument with an initial component of `~` or `~user` replaced by that *user*’s home directory.

On Unix, an initial `~` is replaced by the environment variable `HOME` if it is set; otherwise the current user’s home directory is looked up in the password directory through the built-in module `pwd`. An initial `~user` is looked up directly in the password directory.

On Windows, `HOME` and `USERPROFILE` will be used if set, otherwise a combination of `HOMEPATH` and `HOMEDRIVE` will be used. An initial `~user` is handled by stripping the last directory component from the created user path derived above.

If the expansion fails or if the path does not begin with a tilde, the path is returned unchanged.

**os.path.expandvars(path)**

Return the argument with environment variables expanded. Substrings of the form `$name` or `${name}` are replaced by the value of environment variable *name*. Malformed variable names and references to non-existing variables are left unchanged.

On Windows, `%name%` expansions are supported in addition to `$name` and `${name}`.

**os.path.getatime(path)**

Return the time of last access of *path*. The return value is a number giving the number of seconds since the epoch (see the `time` module). Raise `os.error` if the file does not exist or is
inaccessible.

*New in version 1.5.2.*

*Changed in version 2.3:* If `os.stat_float_times()` returns `True`, the result is a floating point number.

**os.path.getmtime(path)**

Return the time of last modification of `path`. The return value is a number giving the number of seconds since the epoch (see the `time` module). Raise `os.error` if the file does not exist or is inaccessible.

*New in version 1.5.2.*

*Changed in version 2.3:* If `os.stat_float_times()` returns `True`, the result is a floating point number.

**os.path.getctime(path)**

Return the system's ctime which, on some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time for `path`. The return value is a number giving the number of seconds since the epoch (see the `time` module). Raise `os.error` if the file does not exist or is inaccessible.

*New in version 2.3.*

**os.path.getsize(path)**

Return the size, in bytes, of `path`. Raise `os.error` if the file does not exist or is inaccessible.

*New in version 1.5.2.*

**os.path.isabs(path)**

Return `True` if `path` is an absolute pathname. On Unix, that
means it begins with a slash, on Windows that it begins with a (back)slash after chopping off a potential drive letter.

`os.path.isfile(path)`
Return `True` if `path` is an existing regular file. This follows symbolic links, so both `islink()` and `isfile()` can be true for the same path.

`os.path.isdir(path)`
Return `True` if `path` is an existing directory. This follows symbolic links, so both `islink()` and `isdir()` can be true for the same path.

`os.path.islink(path)`
Return `True` if `path` refers to a directory entry that is a symbolic link. Always `False` if symbolic links are not supported by the Python runtime.

`os.path.ismount(path)`
Return `True` if pathname `path` is a mount point: a point in a file system where a different file system has been mounted. The function checks whether `path`’s parent, `path/..`, is on a different device than `path`, or whether `path/..` and `path` point to the same i-node on the same device — this should detect mount points for all Unix and POSIX variants.

`os.path.join(path, *paths)`
Join one or more path components intelligently. The return value is the concatenation of `path` and any members of `*paths` with exactly one directory separator (`os.sep`) following each non-empty part except the last, meaning that the result will only end in a separator if the last part is empty. If a component is an absolute path, all previous components are thrown away and joining continues from the absolute path component.
On Windows, the drive letter is not reset when an absolute path component (e.g., \r'foo') is encountered. If a component contains a drive letter, all previous components are thrown away and the drive letter is reset. Note that since there is a current directory for each drive, os.path.join("c:\", "foo") represents a path relative to the current directory on drive C: (c:foo), not c:foo.

**os.path.normcase(path)**

Normalize the case of a pathname. On Unix and Mac OS X, this returns the path unchanged; on case-insensitive filesystems, it converts the path to lowercase. On Windows, it also converts forward slashes to backward slashes.

**os.path.normpath(path)**

Normalize a pathname by collapsing redundant separators and up-level references so that A//B, A/B/, A/..B and A/foo/../B all become A/B. This string manipulation may change the meaning of a path that contains symbolic links. On Windows, it converts forward slashes to backward slashes. To normalize case, use normcase().

**os.path.realpath(path)**

Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path (if they are supported by the operating system).

*New in version 2.2.*

**os.path.relpath(path[, start])**

Return a relative filepath to path either from the current directory or from an optional start directory. This is a path computation: the filesystem is not accessed to confirm the existence or nature of path or start.
start defaults to \texttt{os.getcwd}.

Availability: Windows, Unix.

New in version 2.6.

\begin{verbatim}
\texttt{os.path\_samefile(path1, path2)}
\end{verbatim}

Return \texttt{True} if both pathname arguments refer to the same file or
directory (as indicated by device number and i-node number). Raise an exception if an \texttt{os.stat()} call on either pathname fails.

Availability: Unix.

\begin{verbatim}
\texttt{os.path\_sameopenfile(fp1, fp2)}
\end{verbatim}

Return \texttt{True} if the file descriptors \texttt{fp1} and \texttt{fp2} refer to the same
file.

Availability: Unix.

\begin{verbatim}
\texttt{os.path\_samestat(stat1, stat2)}
\end{verbatim}

Return \texttt{True} if the stat tuples \texttt{stat1} and \texttt{stat2} refer to the same
file. These structures may have been returned by \texttt{os.fstat()}, \texttt{os.lstat()}, or \texttt{os.stat()}. This function implements the
underlying comparison used by \texttt{samefile()} and \texttt{sameopenfile()}.

Availability: Unix.

\begin{verbatim}
\texttt{os.path\_split(path)}
\end{verbatim}

Split the pathname \texttt{path} into a pair, (\texttt{head}, \texttt{tail}) where \texttt{tail} is the
last pathname component and \texttt{head} is everything leading up to
that. The \texttt{tail} part will never contain a slash; if \texttt{path} ends in a
slash, \texttt{tail} will be empty. If there is no slash in \texttt{path}, \texttt{head} will be
empty. If \texttt{path} is empty, both \texttt{head} and \texttt{tail} are empty. Trailing
slashes are stripped from \texttt{head} unless it is the root (one or more
slashes only). In all cases, \texttt{join(head, tail)} returns a path to the
same location as path (but the strings may differ). Also see the functions dirname() and basename().

os.path.splitdrive(path)
Split the pathname path into a pair (drive, tail) where drive is either a drive specification or the empty string. On systems which do not use drive specifications, drive will always be the empty string. In all cases, drive + tail will be the same as path.

New in version 1.3.

os.path.splitext(path)
Split the pathname path into a pair (root, ext) such that root + ext == path, and ext is empty or begins with a period and contains at most one period. Leading periods on the basename are ignored; splitext('.cshrc') returns ('.cshrc', '')

Changed in version 2.6: Earlier versions could produce an empty root when the only period was the first character.

os.path.splitunc(path)
Split the pathname path into a pair (unc, rest) so that unc is the UNC mount point (such as r'\host\mount'), if present, and rest the rest of the path (such as r'\path\file.ext'). For paths containing drive letters, unc will always be the empty string.

Availability: Windows.

os.path.walk(path, visit, arg)
Calls the function visit with arguments (arg, dirname, names) for each directory in the directory tree rooted at path (including path itself, if it is a directory). The argument dirname specifies the visited directory, the argument names lists the files in the directory (gotten from os.listdir(dirname)). The visit function
may modify names to influence the set of directories visited below dirname, e.g. to avoid visiting certain parts of the tree. (The object referred to by names must be modified in place, using del or slice assignment.)

**Note:** Symbolic links to directories are not treated as subdirectories, and that walk() therefore will not visit them. To visit linked directories you must identify them with os.path.islink(file) and os.path.isdir(file), and invoke walk() as necessary.

**Note:** This function is deprecated and has been removed in Python 3 in favor of os.walk().

os.path.supports_unicode_filenames
  True if arbitrary Unicode strings can be used as file names (within limitations imposed by the file system).

*New in version 2.3.*
10.2. fileinput — Iterate over lines from multiple input streams

Source code: Lib/fileinput.py

This module implements a helper class and functions to quickly write a loop over standard input or a list of files. If you just want to read or write one file see open().

The typical use is:

```python
import fileinput
for line in fileinput.input():
    process(line)
```

This iterates over the lines of all files listed in `sys.argv[1:]`, defaulting to `sys.stdin` if the list is empty. If a filename is '-', it is also replaced by `sys.stdin`. To specify an alternative list of filenames, pass it as the first argument to `input()`. A single file name is also allowed.

All files are opened in text mode by default, but you can override this by specifying the `mode` parameter in the call to `input()` or `FileInput()`. If an I/O error occurs during opening or reading a file, `IOError` is raised.

If `sys.stdin` is used more than once, the second and further use will return no lines, except perhaps for interactive use, or if it has been explicitly reset (e.g. using `sys.stdin.seek(0)`).

Empty files are opened and immediately closed; the only time their presence in the list of filenames is noticeable at all is when the last
file opened is empty.

Lines are returned with any newlines intact, which means that the last line in a file may not have one.

You can control how files are opened by providing an opening hook via the `openhook` parameter to `fileinput.input()` or `FileInput()`. The hook must be a function that takes two arguments, `filename` and `mode`, and returns an accordingly opened file-like object. Two useful hooks are already provided by this module.

The following function is the primary interface of this module:

```python
fileinput.input([files, inplace, backup, bufsize, mode, openhook])
```

Create an instance of the `FileInput` class. The instance will be used as global state for the functions of this module, and is also returned to use during iteration. The parameters to this function will be passed along to the constructor of the `FileInput` class.

*Changed in version 2.5:* Added the `mode` and `openhook` parameters.

*Changed in version 2.7.12:* The `bufsize` parameter is no longer used.

The following functions use the global state created by `fileinput.input()`; if there is no active state, `RuntimeError` is raised.

```python
fileinput.filename()
```

Return the name of the file currently being read. Before the first line has been read, returns `None`.

```python
fileinput.fileno()
```

Return the integer “file descriptor” for the current file. When no
file is opened (before the first line and between files), returns -1.

New in version 2.5.

fileinput.\texttt{lineno}()
Return the cumulative line number of the line that has just been read. Before the first line has been read, returns 0. After the last line of the last file has been read, returns the line number of that line.

fileinput.\texttt{filelineno}()
Return the line number in the current file. Before the first line has been read, returns 0. After the last line of the last file has been read, returns the line number of that line within the file.

fileinput.\texttt{isfirstline}()
Returns true if the line just read is the first line of its file, otherwise returns false.

fileinput.\texttt{isstdin}()
Returns true if the last line was read from \texttt{sys.stdin}, otherwise returns false.

fileinput.\texttt{nextfile}()
Close the current file so that the next iteration will read the first line from the next file (if any); lines not read from the file will not count towards the cumulative line count. The filename is not changed until after the first line of the next file has been read. Before the first line has been read, this function has no effect; it cannot be used to skip the first file. After the last line of the last file has been read, this function has no effect.

fileinput.\texttt{close}()
Close the sequence.
The class which implements the sequence behavior provided by the module is available for subclassing as well:

```python
class fileinput.FileInput([files[, inplace[, backup[, bufsize[, mode[, openhook]]]]]])
```

Class `FileInput` is the implementation; its methods `filename()`, `fileno()`, `lineno()`, `fileleno()`, `isfirstline()`, `isstdin()`, `nextfile()` and `close()` correspond to the functions of the same name in the module. In addition it has a `readline()` method which returns the next input line, and a `__getitem__()` method which implements the sequence behavior. The sequence must be accessed in strictly sequential order; random access and `readline()` cannot be mixed.

With `mode` you can specify which file mode will be passed to `open()`. It must be one of `'r'`, `'rU'`, `'U'` and `'rb'`.

The `openhook`, when given, must be a function that takes two arguments, `filename` and `mode`, and returns an accordingly opened file-like object. You cannot use `inplace` and `openhook` together.

*Changed in version 2.5:* Added the `mode` and `openhook` parameters.

*Changed in version 2.7.12:* The `bufsize` parameter is no longer used.

**Optional in-place filtering:** if the keyword argument `inplace=1` is passed to `fileinput.input()` or to the `FileInput` constructor, the file is moved to a backup file and standard output is directed to the input file (if a file of the same name as the backup file already exists, it will be replaced silently). This makes it possible to write a filter that rewrites its input file in place. If the `backup` parameter is given
(typically as `backup='.<some extension>'`), it specifies the extension for the backup file, and the backup file remains around; by default, the extension is `.bak` and it is deleted when the output file is closed. In-place filtering is disabled when standard input is read.

**Note:** The current implementation does not work for MS-DOS 8+3 filesystems.

The two following opening hooks are provided by this module:

**fileinput.hook_compressed(filename, mode)**

Transparencyly opens files compressed with gzip and bzip2 (recognized by the extensions `.gz` and `.bz2`) using the `gzip` and `bzip2` modules. If the filename extension is not `.gz` or `.bz2`, the file is opened normally (ie, using `open()` without any decompression).

Usage example:

```python
fi = fileinput.FileInput(openhook=fileinput.hook_compressed)
```

*New in version 2.5.*

**fileinput.hook_encoded(encoding)**

Returns a hook which opens each file with `io.open()`, using the given `encoding` to read the file.

Usage example:

```python
fi = fileinput.FileInput(openhook=fileinput.hook_encoded("iso-8859-1"))
```

**Note:** With this hook, `FileInput` might return Unicode strings depending on the specified `encoding`.

*New in version 2.5.*
The `stat` module defines constants and functions for interpreting the results of `os.stat()`, `os.fstat()` and `os.lstat()` (if they exist). For complete details about the `stat()`, `fstat()` and `lstat()` calls, consult the documentation for your system.

The `stat` module defines the following functions to test for specific file types:

- `stat.S_ISDIR(mode)`
  - Return non-zero if the mode is from a directory.

- `stat.S_ISCHR(mode)`
  - Return non-zero if the mode is from a character special device file.

- `stat.S_ISBLK(mode)`
  - Return non-zero if the mode is from a block special device file.

- `stat.S_ISREG(mode)`
  - Return non-zero if the mode is from a regular file.

- `stat.S_ISFIFO(mode)`
  - Return non-zero if the mode is from a FIFO (named pipe).

- `stat.S_ISLNK(mode)`
  - Return non-zero if the mode is from a symbolic link.

- `stat.S_ISSOCK(mode)`
Return non-zero if the mode is from a socket.

Two additional functions are defined for more general manipulation of the file's mode:

```
stat.S_IMODE(mode)

Return the portion of the file's mode that can be set by os.chmod()—that is, the file's permission bits, plus the sticky bit, set-group-id, and set-user-id bits (on systems that support them).
```

```
stat.S_IFMT(mode)

Return the portion of the file's mode that describes the file type (used by the s_IS*() functions above).
```

Normally, you would use the os.path.is*() functions for testing the type of a file; the functions here are useful when you are doing multiple tests of the same file and wish to avoid the overhead of the stat() system call for each test. These are also useful when checking for information about a file that isn't handled by os.path, like the tests for block and character devices.

Example:

```python
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top, calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.stat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
```
else:
    # Unknown file type, print a message
    print 'Skipping %s' % pathname

def visitfile(file):
    print 'visiting', file

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)

All the variables below are simply symbolic indexes into the 10-tuple returned by os.stat(), os.fstat() or os.lstat().

stat.ST_MODE
    Inode protection mode.

stat.ST_INO
    Inode number.

stat.ST_DEV
    Device inode resides on.

stat.ST_NLINK
    Number of links to the inode.

stat.ST_UID
    User id of the owner.

stat.ST_GID
    Group id of the owner.

stat.ST_SIZE
    Size in bytes of a plain file; amount of data waiting on some special files.

stat.ST_ATIME
    Time of last access.

stat.ST_MTIME
Time of last modification.

**stat.ST_CTIME**

The “ctime” as reported by the operating system. On some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time (see platform documentation for details).

The interpretation of “file size” changes according to the file type. For plain files this is the size of the file in bytes. For FIFOs and sockets under most flavors of Unix (including Linux in particular), the “size” is the number of bytes waiting to be read at the time of the call to `os.stat()`, `os.fstat()`, or `os.lstat()`; this can sometimes be useful, especially for polling one of these special files after a non-blocking open. The meaning of the size field for other character and block devices varies more, depending on the implementation of the underlying system call.

The variables below define the flags used in the **ST_MODE** field.

Use of the functions above is more portable than use of the first set of flags:

**stat.S_IFSOCK**

Socket.

**stat.S_IFLNK**

Symbolic link.

**stat.S_IFREG**

Regular file.

**stat.S_IFBLK**

Block device.

**stat.S_IFDIR**
Directory.

`stat.S_IFCHR`
Character device.

`stat.S_IFIFO`
FIFO.

The following flags can also be used in the `mode` argument of `os.chmod()`:

`stat.S_ISUID`
Set UID bit.

`stat.S_ISGID`
Set-group-ID bit. This bit has several special uses. For a directory it indicates that BSD semantics is to be used for that directory: files created there inherit their group ID from the directory, not from the effective group ID of the creating process, and directories created there will also get the `S_ISGID` bit set. For a file that does not have the group execution bit (`S_IXGRP`) set, the set-group-ID bit indicates mandatory file/record locking (see also `S_ENFMT`).

`stat.S_ISVTX`
Sticky bit. When this bit is set on a directory it means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, or by a privileged process.

`stat.S_IRWXU`
Mask for file owner permissions.

`stat.S_IRUSR`
Owner has read permission.

`stat.S_IWUSR`
Owner has write permission.

stat. **S_IXUSR**
Owner has execute permission.

stat. **S_IRWXG**
Mask for group permissions.

stat. **S_IRGRP**
Group has read permission.

stat. **S_IWGRP**
Group has write permission.

stat. **S_IXGRP**
Group has execute permission.

stat. **S_IRWIXO**
Mask for permissions for others (not in group).

stat. **S_IROTH**
Others have read permission.

stat. **S_IWOTH**
Others have write permission.

stat. **S_IXOTH**
Others have execute permission.

stat. **S_ENFMT**
System V file locking enforcement. This flag is shared with
**S_ISGID**: file/record locking is enforced on files that do not have
the group execution bit (**S_IXGRP**) set.

stat. **S_IREAD**
Unix V7 synonym for **S_IRUSR**.

stat. **S_IWRITE**
Unix V7 synonym for S_IWUSR.

stat. S_IEXEC
Unix V7 synonym for S_IXUSR.

The following flags can be used in the flags argument of os.chflags():

stat. UF_NODUMP
   Do not dump the file.

stat. UF_IMMUTABLE
   The file may not be changed.

stat. UF_APPEND
   The file may only be appended to.

stat. UF_OPAQUE
   The directory is opaque when viewed through a union stack.

stat. UF_NOUNLINK
   The file may not be renamed or deleted.

stat. UF_COMPRESSED
   The file is stored compressed (Mac OS X 10.6+).

stat. UF_HIDDEN
   The file should not be displayed in a GUI (Mac OS X 10.5+).

stat. SF_ARCHIVED
   The file may be archived.

stat. SF_IMMUTABLE
   The file may not be changed.

stat. SF_APPEND
   The file may only be appended to.
stat. **SF_NOUNLINK**

The file may not be renamed or deleted.

stat. **SF_SNAPSHOT**

The file is a snapshot file.

See the *BSD or Mac OS systems man page *chflags(2)* for more information.
10.4. `statvfs` — Constants used with `os.statvfs()`

Deprecated since version 2.6: The `statvfs` module has been removed in Python 3.

The `statvfs` module defines constants so interpreting the result of `os.statvfs()`, which returns a tuple, can be made without remembering “magic numbers.” Each of the constants defined in this module is the *index* of the entry in the tuple returned by `os.statvfs()` that contains the specified information.

`statvfs.F_BSIZE`

Preferred file system block size.

`statvfs.F_FRSIZE`

Fundamental file system block size.

`statvfs.F_BLOCKS`

Total number of blocks in the filesystem.

`statvfs.F_BFREE`

Total number of free blocks.

`statvfs.F_BAVAIL`

Free blocks available to non-super user.

`statvfs.F_FILES`

Total number of file nodes.

`statvfs.F_FFREE`

Total number of free file nodes.

`statvfs.F_FAVAIL`
Free nodes available to non-super user.

```
statvfs.F_FLAG
    Flags. System dependent: see statvfs() man page.
```

```
statvfs.F_NAME_MAX
    Maximum file name length.
```
10.5. filecmp — File and Directory Comparisons

Source code: Lib/filecmp.py

The `filecmp` module defines functions to compare files and directories, with various optional time/correctness trade-offs. For comparing files, see also the `difflib` module.

The `filecmp` module defines the following functions:

```python
filecmp.cmp(f1, f2[, shallow])
```

Compare the files named `f1` and `f2`, returning `True` if they seem equal, `False` otherwise.

Unless `shallow` is given and is false, files with identical `os.stat()` signatures are taken to be equal.

Files that were compared using this function will not be compared again unless their `os.stat()` signature changes.

Note that no external programs are called from this function, giving it portability and efficiency.

```python
filecmp.cmpfiles(dir1, dir2, common[, shallow])
```

Compare the files in the two directories `dir1` and `dir2` whose names are given by `common`.

Returns three lists of file names: `match`, `mismatch`, `errors`. `match` contains the list of files that match, `mismatch` contains the names of those that don’t, and `errors` lists the names of files which could not be compared. Files are listed in `errors` if they don’t exist in
one of the directories, the user lacks permission to read them or if the comparison could not be done for some other reason.

The *shallow* parameter has the same meaning and default value as for *filecmp.cmp()*.

For example, `cmpfiles('a', 'b', ['c', 'd/e'])` will compare `a/c` with `b/c` and `a/d/e` with `b/d/e`. `'c'` and `d/e'` will each be in one of the three returned lists.

Example:

```python
>>> import filecmp
>>> filecmp.cmp('undoc.rst', 'undoc.rst')
True
>>> filecmp.cmp('undoc.rst', 'index.rst')
False
```
10.5.1. The dircmp class

dircmp instances are built using this constructor:

class filecmp.dircmp(a, b[, ignore[, hide]])

Construct a new directory comparison object, to compare the directories a and b. ignore is a list of names to ignore, and defaults to ['RCS', 'CVS', 'tags']. hide is a list of names to hide, and defaults to [os.curdir, os.pardir].

The dircmp class compares files by doing shallow comparisons as described for filecmp.cmp().

The dircmp class provides the following methods:

report()
Print (to sys.stdout) a comparison between a and b.

report_partial_closure()
Print a comparison between a and b and common immediate subdirectories.

report_full_closure()
Print a comparison between a and b and common subdirectories (recursively).

The dircmp class offers a number of interesting attributes that may be used to get various bits of information about the directory trees being compared.

Note that via __getattr__() hooks, all attributes are computed lazily, so there is no speed penalty if only those attributes which are lightweight to compute are used.
The directory *a*.

The directory *b*.

Files and subdirectories in *a*, filtered by *hide* and *ignore*.

Files and subdirectories in *b*, filtered by *hide* and *ignore*.

Files and subdirectories in both *a* and *b*.

Files and subdirectories only in *a*.

Files and subdirectories only in *b*.

Subdirectories in both *a* and *b*.

Files in both *a* and *b*

Names in both *a* and *b*, such that the type differs between the directories, or names for which *os.stat()* reports an error.

Files which are identical in both *a* and *b*, using the class's file comparison operator.

Files which are in both *a* and *b*, whose contents differ
according to the class’s file comparison operator.

funny_files

Files which are in both a and b, but could not be compared.

subdirs

A dictionary mapping names in common_dirs to dircmp objects.

Here is a simplified example of using the subdirs attribute to search recursively through two directories to show common different files:

```python
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...   for name in dcmp.diff_files:
...     print "diff_file %s found in %s and %s" % (name, dcmp.left, dcmp.right)
...   for sub_dcmp in dcmp.subdirs.values():
...     print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```
This module generates temporary files and directories. It works on all supported platforms.

In version 2.3 of Python, this module was overhauled for enhanced security. It now provides three new functions, `NamedTemporaryFile()`, `mkstemp()`, and `mkdtemp()`, which should eliminate all remaining need to use the insecure `mktemp()` function. Temporary file names created by this module no longer contain the process ID; instead a string of six random characters is used.

Also, all the user-callable functions now take additional arguments which allow direct control over the location and name of temporary files. It is no longer necessary to use the global `tempdir` and `template` variables. To maintain backward compatibility, the argument order is somewhat odd; it is recommended to use keyword arguments for clarity.

The module defines the following user-callable functions:

```python
tempfile.TemporaryFile([mode='w+b', bufsize=-1, suffix='', prefix='tmp', dir=None])
```

Return a file-like object that can be used as a temporary storage area. The file is created using `mkstemp()`. It will be destroyed as soon as it is closed (including an implicit close when the object is garbage collected). Under Unix, the directory entry for the file is removed immediately after the file is created. Other platforms do not support this; your code should not rely on a temporary file
created using this function having or not having a visible name in the file system.

The *mode* parameter defaults to `'w+b'` so that the file created can be read and written without being closed. Binary mode is used so that it behaves consistently on all platforms without regard for the data that is stored. *bufsize* defaults to `-1`, meaning that the operating system default is used.

The *dir*, *prefix* and *suffix* parameters are passed to `mkstemp()`.

The returned object is a true file object on POSIX platforms. On other platforms, it is a file-like object whose *file* attribute is the underlying true file object. This file-like object can be used in a *with* statement, just like a normal file.

```python
tempfile.NamedTemporaryFile([mode='w+b', bufsize=-1, suffix='', prefix='tmp', dir=None, delete=True])
```

This function operates exactly as `TemporaryFile()` does, except that the file is guaranteed to have a visible name in the file system (on Unix, the directory entry is not unlinked). That name can be retrieved from the *name* attribute of the returned file-like object. Whether the name can be used to open the file a second time, while the named temporary file is still open, varies across platforms (it can be so used on Unix; it cannot on Windows NT or later). If *delete* is true (the default), the file is deleted as soon as it is closed.

The returned object is always a file-like object whose *file* attribute is the underlying true file object. This file-like object can be used in a *with* statement, just like a normal file.

*New in version 2.3.*
New in version 2.6: The delete parameter.

tempfile.SpooledTemporaryFile([max_size=0, mode='w+b', bufsize=-1, suffix='', prefix='tmp', dir=None])

This function operates exactly as TemporaryFile() does, except that data is spooled in memory until the file size exceeds max_size, or until the file’s fileno() method is called, at which point the contents are written to disk and operation proceeds as with TemporaryFile(). Also, it's truncate method does not accept a size argument.

The resulting file has one additional method, rollover(), which causes the file to roll over to an on-disk file regardless of its size.

The returned object is a file-like object whose _file attribute is either a StringIO object or a true file object, depending on whether rollover() has been called. This file-like object can be used in a with statement, just like a normal file.

New in version 2.6.

        tempfile.mkstemp([suffix='', prefix='tmp', dir=None, text=False])

Creates a temporary file in the most secure manner possible. There are no race conditions in the file’s creation, assuming that the platform properly implements the os.O_EXCL flag for os.open(). The file is readable and writable only by the creating user ID. If the platform uses permission bits to indicate whether a file is executable, the file is executable by no one. The file descriptor is not inherited by child processes.

Unlike TemporaryFile(), the user of mkstemp() is responsible for deleting the temporary file when done with it.
If *suffix* is specified, the file name will end with that suffix, otherwise there will be no suffix. `mkstemp()` does not put a dot between the file name and the suffix; if you need one, put it at the beginning of *suffix*.

If *prefix* is specified, the file name will begin with that prefix; otherwise, a default prefix is used.

If *dir* is specified, the file will be created in that directory; otherwise, a default directory is used. The default directory is chosen from a platform-dependent list, but the user of the application can control the directory location by setting the `TMPDIR`, `TEMP` or `TMP` environment variables. There is thus no guarantee that the generated filename will have any nice properties, such as not requiring quoting when passed to external commands via `os.popen()`.

If *text* is specified, it indicates whether to open the file in binary mode (the default) or text mode. On some platforms, this makes no difference.

`mkstemp()` returns a tuple containing an OS-level handle to an open file (as would be returned by `os.open()`) and the absolute pathname of that file, in that order.

New in version 2.3.

tempfile.``mkdtemp([suffix='', prefix='tmp', dir=None])``

Creates a temporary directory in the most secure manner possible. There are no race conditions in the directory's creation. The directory is readable, writable, and searchable only by the creating user ID.

The user of `mkdtemp()` is responsible for deleting the temporary directory and its contents when done with it.
The *prefix*, *suffix*, and *dir* arguments are the same as for `mkstemp()`.

`mkdtemp()` returns the absolute pathname of the new directory.

*New in version 2.3.*

```python
tempfile.mktemp([suffix='', prefix='tmp', dir=None])
```

*Deprecated since version 2.3: Use `mkstemp()` instead.*

Return an absolute pathname of a file that did not exist at the time the call is made. The *prefix*, *suffix*, and *dir* arguments are the same as for `mkstemp()`.

**Warning:** Use of this function may introduce a security hole in your program. By the time you get around to doing anything with the file name it returns, someone else may have beaten you to the punch. `mktemp()` usage can be replaced easily with `NamedTemporaryFile()`, passing it the `delete=False` parameter:

```python
>>> f = NamedTemporaryFile(delete=False)
>>> f
<open file '<fdopen>', mode 'w+b' at 0x384698>
>>> f.name
'/var/folders/5q/5qTPn6xq2RaWqk+1Ytw3-U+++TI/-Tmp/-tmpG7V1`
>>> f.write("Hello World!\n")
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

The module uses a global variable that tell it how to construct a temporary name. They are initialized at the first call to any of the functions above. The caller may change them, but this is discouraged; use the appropriate function arguments, instead.
tempfile.tempdir

When set to a value other than None, this variable defines the default value for the dir argument to all the functions defined in this module.

If tempfile.tempdir is unset or None at any call to any of the above functions, Python searches a standard list of directories and sets tempfile.tempdir to the first one which the calling user can create files in. The list is:

1. The directory named by the TMPDIR environment variable.
2. The directory named by the TEMP environment variable.
3. The directory named by the TMP environment variable.
4. A platform-specific location:
   ○ On RiscOS, the directory named by the Wimp$ScrapDir environment variable.
   ○ On Windows, the directories C:\TEMP, C:\TMP, \TEMP, and \TMP, in that order.
   ○ On all other platforms, the directories /tmp, /var/tmp, and /usr/tmp, in that order.
5. As a last resort, the current working directory.

tempfile.gettempdir()

Return the directory currently selected to create temporary files in. If tempfile.tempdir is not None, this simply returns its contents; otherwise, the search described above is performed, and the result returned.

New in version 2.3.

tempfile.template

Deprecated since version 2.0: Use gettempprefix() instead.

When set to a value other than None, this variable defines the
prefix of the final component of the filenames returned by \texttt{mktemp()}. A string of six random letters and digits is appended to the prefix to make the filename unique. The default prefix is \texttt{tmp}.

Older versions of this module used to require that \texttt{template} be set to \texttt{None} after a call to \texttt{os.fork()}; this has not been necessary since version 1.5.2.

\texttt{tempfile.gettempprefix()}

Return the filename prefix used to create temporary files. This does not contain the directory component. Using this function is preferred over reading the \texttt{template} variable directly.

\textit{New in version 1.5.2.}
10.7. `glob` — Unix style pathname pattern expansion

**Source code:** Lib/glob.py

The `glob` module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order. No tilde expansion is done, but *, ?, and character ranges expressed with [] will be correctly matched. This is done by using the `os.listdir()` and `fnmatch.fnmatch()` functions in concert, and not by actually invoking a subshell. Note that unlike `fnmatch.fnmatch()`, `glob` treats filenames beginning with a dot (.) as special cases. (For tilde and shell variable expansion, use `os.path.expanduser()` and `os.path.expandvars()`.)

For a literal match, wrap the meta-characters in brackets. For example, `'[?]'` matches the character `?`.

`glob.glob(pathname)`

Return a possibly-empty list of path names that match `pathname`, which must be a string containing a path specification. `pathname` can be either absolute (like `/usr/src/Python-1.5/Makefile`) or relative (like `../../Tools/*//*.gif`), and can contain shell-style wildcards. Broken symlinks are included in the results (as in the shell).

`glob.iglob(pathname)`

Return an iterator which yields the same values as `glob()` without actually storing them all simultaneously.
New in version 2.5.

For example, consider a directory containing only the following files: 1.gif, 2.txt, and card.gif. `glob()` will produce the following results. Notice how any leading components of the path are preserved.

```python
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?\.gif')
['1.gif']
```

If the directory contains files starting with . they won’t be matched by default. For example, consider a directory containing card.gif and .card.gif:

```python
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

See also:

Module `fnmatch`

Shell-style filename (not path) expansion
This module provides support for Unix shell-style wildcards, which are not the same as regular expressions (which are documented in the `re` module). The special characters used in shell-style wildcards are:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>matches everything</td>
</tr>
<tr>
<td>?</td>
<td>matches any single character</td>
</tr>
<tr>
<td>[seq]</td>
<td>matches any character in <code>seq</code></td>
</tr>
<tr>
<td>[!seq]</td>
<td>matches any character not in <code>seq</code></td>
</tr>
</tbody>
</table>

For a literal match, wrap the meta-characters in brackets. For example, `'[?]'` matches the character `'?'`.

Note that the filename separator (`'/` on Unix) is not special to this module. See module `glob` for pathname expansion (`
` uses `fnmatch()` to match pathname segments). Similarly, filenames starting with a period are not special for this module, and are matched by the `*` and `?` patterns.

```
fnmatch.fnmatch(filename, pattern)
```

Test whether the `filename` string matches the `pattern` string, returning `True` or `False`. If the operating system is case-insensitive, then both parameters will be normalized to all lower- or upper-case before the comparison is performed. `fnmatchcase()` can be used to perform a case-sensitive
comparison, regardless of whether that’s standard for the operating system.

This example will print all file names in the current directory with the extension .txt:

```python
import fnmatch
import os

for file in os.listdir('.'):  
    if fnmatch.fnmatch(file, '*.txt'):
        print file
```

*fnmatch.fnmatchcase(filename, pattern)*

Test whether `filename` matches `pattern`, returning `True` or `False`; the comparison is case-sensitive.

*fnmatch.filter(names, pattern)*

Return the subset of the list of `names` that match `pattern`. It is the same as `[n for n in names if fnmatch(n, pattern)]`, but implemented more efficiently.

*New in version 2.2.*

*fnmatch.translate(pattern)*

Return the shell-style `pattern` converted to a regular expression for using with `re.match()`.

Example:

```python
>>> import fnmatch, re
>>> 
>>> regex = fnmatch.translate('*.txt')
>>> regex
'.*\.txt\Z(?ms)' 
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<_sre.SRE_Match object at 0x...>
```
See also:

**Module glob**

Unix shell-style path expansion.
10.9. linecache — Random access to text lines

Source code: Lib/linecache.py

The linecache module allows one to get any line from any file, while attempting to optimize internally, using a cache, the common case where many lines are read from a single file. This is used by the traceback module to retrieve source lines for inclusion in the formatted traceback.

The linecache module defines the following functions:

linecache.getline(filename, lineno[, module_globals])
Get line lineno from file named filename. This function will never raise an exception — it will return '' on errors (the terminating newline character will be included for lines that are found).

If a file named filename is not found, the function will look for it in the module search path, sys.path, after first checking for a PEP 302 __loader__ in module_globals, in case the module was imported from a zipfile or other non-filesystem import source.

New in version 2.5: The module_globals parameter was added.

linecache.clearcache()
Clear the cache. Use this function if you no longer need lines from files previously read using getline().

linecache.checkcache([filename])
Check the cache for validity. Use this function if files in the cache
may have changed on disk, and you require the updated version. If filename is omitted, it will check all the entries in the cache.

Example:

```python
>>> import linecache
>>> linecache.getline('/etc/passwd', 4)
'sys:x:3:3:sys:/dev:/bin/sh
'n'
```
The `shutil` module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal. For operations on individual files, see also the `os` module.

**Warning:** Even the higher-level file copying functions (`shutil.copy()`, `shutil.copy2()`) can’t copy all file metadata.

On POSIX platforms, this means that file owner and group are lost as well as ACLs. On Mac OS, the resource fork and other metadata are not used. This means that resources will be lost and file type and creator codes will not be correct. On Windows, file owners, ACLs and alternate data streams are not copied.
10.10.1. Directory and files operations

`shutil.copyfileobj(fsrc, fdst[, length])`
Copy the contents of the file-like object `fsrc` to the file-like object `fdst`. The integer `length`, if given, is the buffer size. In particular, a negative `length` value means to copy the data without looping over the source data in chunks; by default the data is read in chunks to avoid uncontrolled memory consumption. Note that if the current file position of the `fsrc` object is not 0, only the contents from the current file position to the end of the file will be copied.

`shutil.copyfile(src, dst)`
Copy the contents (no metadata) of the file named `src` to a file named `dst`. `dst` must be the complete target file name; look at `shutil.copy()` for a copy that accepts a target directory path. If `src` and `dst` are the same files, `Error` is raised. The destination location must be writable; otherwise, an `IOError` exception will be raised. If `dst` already exists, it will be replaced. Special files such as character or block devices and pipes cannot be copied with this function. `src` and `dst` are path names given as strings.

`shutil.copymode(src, dst)`
Copy the permission bits from `src` to `dst`. The file contents, owner, and group are unaffected. `src` and `dst` are path names given as strings.

`shutil.copystat(src, dst)`
Copy the permission bits, last access time, last modification time, and flags from `src` to `dst`. The file contents, owner, and group are unaffected. `src` and `dst` are path names given as strings.

`shutil.copy(src, dst)`
Copy the file `src` to the file or directory `dst`. If `dst` is a directory, a file with the same basename as `src` is created (or overwritten) in the directory specified. Permission bits are copied. `src` and `dst` are path names given as strings.

`shutil.copy2(src, dst)`

Similar to `shutil.copy()`, but metadata is copied as well – in fact, this is just `shutil.copy()` followed by `copystat()`. This is similar to the Unix command `cp -p`.

`shutil.ignore_patterns(*patterns)`

This factory function creates a function that can be used as a callable for `copytree()`’s `ignore` argument, ignoring files and directories that match one of the glob-style `patterns` provided. See the example below.

*New in version 2.6.*

`shutil.copytree(src, dst, symlinks=False, ignore=None)`

Recursively copy an entire directory tree rooted at `src`. The destination directory, named by `dst`, must not already exist; it will be created as well as missing parent directories. Permissions and times of directories are copied with `copystat()`, individual files are copied using `shutil.copy2()`.

If `symlinks` is true, symbolic links in the source tree are represented as symbolic links in the new tree, but the metadata of the original links is NOT copied; if false or omitted, the contents and metadata of the linked files are copied to the new tree.

If `ignore` is given, it must be a callable that will receive as its arguments the directory being visited by `copytree()`, and a list of its contents, as returned by `os.listdir()`. Since `copytree()` is called recursively, the `ignore` callable will be called once for each
directory that is copied. The callable must return a sequence of
directory and file names relative to the current directory (i.e. a
subset of the items in its second argument); these names will
then be ignored in the copy process. \texttt{ignore\_patterns()} can be
used to create such a callable that ignores names based on glob-
style patterns.

If exception(s) occur, an \texttt{Error} is raised with a list of reasons.

The source code for this should be considered an example rather
than the ultimate tool.

\textit{Changed in version 2.3:} \texttt{Error} is raised if any exceptions occur
during copying, rather than printing a message.

\textit{Changed in version 2.5:} Create intermediate directories needed
to create \texttt{dst}, rather than raising an error. Copy permissions and
times of directories using \texttt{copystat()}.

\textit{Changed in version 2.6:} Added the \texttt{ignore} argument to be able to
influence what is being copied.

\texttt{shutil.rmtree(path[, ignore\_errors[, onerror]])}
Delete an entire directory tree; \texttt{path} must point to a directory (but
not a symbolic link to a directory). If \texttt{ignore\_errors} is true, errors
resulting from failed removals will be ignored; if false or omitted,
such errors are handled by calling a handler specified by \texttt{onerror}
or, if that is omitted, they raise an exception.

If \texttt{onerror} is provided, it must be a callable that accepts three
parameters: \texttt{function}, \texttt{path}, and \texttt{excinfo}. The first parameter,
\texttt{function}, is the function which raised the exception; it will be
\texttt{os.path.islink()}, \texttt{os.listdir()}, \texttt{os.remove()} or \texttt{os.rmdir()}. The
second parameter, \texttt{path}, will be the path name passed to
\texttt{function}. The third parameter, \texttt{excinfo}, will be the exception
information return by `sys.exc_info()`. Exceptions raised by `onerror` will not be caught.

*Changed in version 2.6:* Explicitly check for `path` being a symbolic link and raise `OSError` in that case.

```python
shutil.move(src, dst)
```

Recursively move a file or directory (`src`) to another location (`dst`).

If the destination is an existing directory, then `src` is moved inside that directory. If the destination already exists but is not a directory, it may be overwritten depending on `os.rename()` semantics.

If the destination is on the current filesystem, then `os.rename()` is used. Otherwise, `src` is copied (using `shutil.copy2()`) to `dst` and then removed.

*New in version 2.3.*

**exception** `shutil.Error`

This exception collects exceptions that are raised during a multi-file operation. For `copytree()`, the exception argument is a list of 3-tuples (`srcname, dstname, exception`).

*New in version 2.3.*

## 10.10.1.1. copytree example

This example is the implementation of the `copytree()` function, described above, with the docstring omitted. It demonstrates many of the other functions provided by this module.

```python
def copytree(src, dst, symlinks=False, ignore=None):
    names = os.listdir(src)
    if ignore is not None:
```
ignored_names = ignore(src, names)
else:
    ignored_names = set()

os.makedirs(dst)
errors = []
for name in names:
    if name in ignored_names:
        continue
    srcname = os.path.join(src, name)
dstname = os.path.join(dst, name)
try:
    if symlinks and os.path.islink(srcname):
        linkto = os.readlink(srcname)
        os.symlink(linkto, dstname)
    elif os.path.isdir(srcname):
        copytree(srcname, dstname, symlinks, ignore)
    else:
        copy2(srcname, dstname)
    # XXX What about devices, sockets etc.?
except (IOError, os.error) as why:
    errors.append(((srcname, dstname, str(why))))
    # catch the Error from the recursive copytree so that we
    # can continue with other files
except Error as err:
    errors.extend(err.args[0])
try:
    copystat(src, dst)
except WindowsError:
    # can't copy file access times on Windows
    pass
except OSError as why:
    errors.extend(((src, dst, str(why)))
if errors:
    raise Error(errors)

Another example that uses the ignore_patterns() helper:

```python
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*', 'other'),...
whose name starts with tmp.

Another example that uses the ignore argument to add a logging call:

```python
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s' % path)
    return [] # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```
10.10.2. Archiving operations

High-level utilities to create and read compressed and archived files are also provided. They rely on the `zipfile` and `tarfile` modules.

```python
shutil.make_archive(base_name, format[, root_dir[, base_dir[, verbose[, dry_run[, owner[, group[, logger]]]]]]])
```

Create an archive file (eg. zip or tar) and returns its name.

- `base_name` is the name of the file to create, including the path, minus any format-specific extension. `format` is the archive format: one of “zip”, “tar”, “bztar” or “gztar”.

- `root_dir` is a directory that will be the root directory of the archive; ie. we typically chdir into `root_dir` before creating the archive.

- `base_dir` is the directory where we start archiving from; ie. `base_dir` will be the common prefix of all files and directories in the archive.

- `root_dir` and `base_dir` both default to the current directory.

- `owner` and `group` are used when creating a tar archive. By default, uses the current owner and group.

- `logger` must be an object compatible with PEP 282, usually an instance of `logging.Logger`.

  `New in version 2.7`

```python
shutil.get_archive_formats()
```

Return a list of supported formats for archiving. Each element of the returned sequence is a tuple `(name, description)`.
By default `shutil` provides these formats:

- `gztar`: gzip'ed tar-file
- `bztar`: bzip2'ed tar-file
- `tar`: uncompressed tar file
- `zip`: ZIP file

You can register new formats or provide your own archiver for any existing formats, by using `register_archive_format()`.

*New in version 2.7.*

```python
shutil.register_archive_format(name, function[, extra_args[, description]])
```

Register an archiver for the format `name`. `function` is a callable that will be used to invoke the archiver.

If given, `extra_args` is a sequence of `(name, value)` that will be used as extra keywords arguments when the archiver callable is used.

`description` is used by `get_archive_formats()` which returns the list of archivers. Defaults to an empty list.

*New in version 2.7.*

```python
shutil.unregister_archive_format(name)
```

Remove the archive format `name` from the list of supported formats.

*New in version 2.7.*

### 10.10.2.1. Archiving example

In this example, we create a gzip'ed tar-file archive containing all
files found in the `.ssh` directory of the user:

```python
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

The resulting archive contains:

```
$ tar -tзвf /Users/tarek/myarchive.tar.gz
```

```
drwx------ tarek/staff 0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff 609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff 65 2008-06-09 13:26:54 ./config
-rwx------ tarek/staff 668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff 609 2008-06-09 13:26:54 ./id_dsa.pub
-rwx------ tarek/staff 1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff 397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff 37192 2010-02-06 18:23:10 ./known_hosts
```
The `dircache` module defines a function for reading directory listing using a cache, and cache invalidation using the `mtime` of the directory. Additionally, it defines a function to annotate directories by appending a slash.

The `dircache` module defines the following functions:

- `dircache.reset()`
  Resets the directory cache.

- `dircache.listdir(path)`
  Return a directory listing of `path`, as gotten from `os.listdir()`. Note that unless `path` changes, further call to `listdir()` will not re-read the directory structure.

  Note that the list returned should be regarded as read-only. (Perhaps a future version should change it to return a tuple?)

- `dircache.opendir(path)`
  Same as `listdir()`. Defined for backwards compatibility.

- `dircache.annotate(head, list)`
  Assume `list` is a list of paths relative to `head`, and append, in place, a `'/ '` to each path which points to a directory.

```python
>>> import dircache
>>> a = dircache.listdir('/')
```
```python
>>> a = a[:]  # Copy the return value so we can change 'a'
>>> a
['bin', 'boot', 'cdrom', 'dev', 'etc', 'floppy', 'home', 'initrd', 'mnt', 'proc', 'root', 'sbin', 'tmp', 'usr', 'var', 'vm

>>> dircache.annotate('/', a)

>>> a
['bin/', 'boot/', 'cdrom/', 'dev/', 'etc/', 'floppy/', 'home/', ', 'lost+found/', 'mnt/', 'proc/', 'root/', 'sbin/', 'tmp/', 'u
linuz']
```
10.12. `macpath` — Mac OS 9 path manipulation functions

This module is the Mac OS 9 (and earlier) implementation of the `os.path` module. It can be used to manipulate old-style Macintosh pathnames on Mac OS X (or any other platform).

The following functions are available in this module: `normcase()`, `normpath()`, `isabs()`, `join()`, `split()`, `isdir()`, `isfile()`, `walk()`, `exists()`. For other functions available in `os.path` dummy counterparts are available.
11. Data Persistence

The modules described in this chapter support storing Python data in a persistent form on disk. The `pickle` and `marshal` modules can turn many Python data types into a stream of bytes and then recreate the objects from the bytes. The various DBM-related modules support a family of hash-based file formats that store a mapping of strings to other strings. The `bsddb` module also provides such disk-based string-to-string mappings based on hashing, and also supports B-Tree and record-based formats.

The list of modules described in this chapter is:

- **11.1. pickle** — Python object serialization
  - 11.1.1. Relationship to other Python modules
  - 11.1.2. Data stream format
  - 11.1.3. Usage
  - 11.1.4. What can be pickled and unpickled?
  - 11.1.5. The pickle protocol
    - 11.1.5.1. Pickling and unpickling normal class instances
    - 11.1.5.2. Pickling and unpickling extension types
    - 11.1.5.3. Pickling and unpickling external objects
  - 11.1.6. Subclassing Unpicklers
  - 11.1.7. Example
- **11.2. cPickle** — A faster `pickle`
- **11.3. copy_reg** — Register `pickle` support functions
  - 11.3.1. Example
- **11.4. shelf** — Python object persistence
  - 11.4.1. Restrictions
  - 11.4.2. Example
- **11.5. marshal** — Internal Python object serialization
- **11.6. anydbm** — Generic access to DBM-style databases
• 11.7. **whichdb** — Guess which DBM module created a database
• 11.8. **dbm** — Simple “database” interface
• 11.9. **gdbm** — GNU's reinterpretation of dbm
• 11.10. **dbhash** — DBM-style interface to the BSD database library
  • 11.10.1. Database Objects
• 11.11. **bsddb** — Interface to Berkeley DB library
  • 11.11.1. Hash, BTree and Record Objects
• 11.12. **dumbdbm** — Portable DBM implementation
  • 11.12.1. Dumbdbm Objects
• 11.13. **sqlite3** — DB-API 2.0 interface for SQLite databases
  • 11.13.1. Module functions and constants
  • 11.13.2. Connection Objects
  • 11.13.3. Cursor Objects
  • 11.13.4. Row Objects
  • 11.13.5. SQLite and Python types
    • 11.13.5.1. Introduction
    • 11.13.5.2. Using adapters to store additional Python types in SQLite databases
      • 11.13.5.2.1. Letting your object adapt itself
      • 11.13.5.2.2. Registering an adapter callable
    • 11.13.5.3. Converting SQLite values to custom Python types
    • 11.13.5.4. Default adapters and converters
  • 11.13.6. Controlling Transactions
  • 11.13.7. Using **sqlite3** efficiently
    • 11.13.7.1. Using shortcut methods
    • 11.13.7.2. Accessing columns by name instead of by index
    • 11.13.7.3. Using the connection as a context manager
  • 11.13.8. Common issues
    • 11.13.8.1. Multithreading
The `pickle` module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as “serialization”, “marshalling,” [1] or “flattening”, however, to avoid confusion, the terms used here are “pickling” and “unpickling”.

This documentation describes both the `pickle` module and the `cPickle` module.

**Warning:** The `pickle` module is not secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.
11.1.1. Relationship to other Python modules

The `pickle` module has an optimized cousin called the `cPickle` module. As its name implies, `cPickle` is written in C, so it can be up to 1000 times faster than `pickle`. However it does not support subclassing of the `Pickler()` and `Unpickler()` classes, because in `cPickle` these are functions, not classes. Most applications have no need for this functionality, and can benefit from the improved performance of `cPickle`. Other than that, the interfaces of the two modules are nearly identical; the common interface is described in this manual and differences are pointed out where necessary. In the following discussions, we use the term “pickle” to collectively describe the `pickle` and `cPickle` modules.

The data streams the two modules produce are guaranteed to be interchangeable.

Python has a more primitive serialization module called `marshal`, but in general `pickle` should always be the preferred way to serialize Python objects. `marshal` exists primarily to support Python’s `.pyc` files.

The `pickle` module differs from `marshal` in several significant ways:

- The `pickle` module keeps track of the objects it has already serialized, so that later references to the same object won’t be serialized again. `marshal` doesn’t do this.

  This has implications both for recursive objects and object sharing. Recursive objects are objects that contain references to themselves. These are not handled by `marshal`, and in fact,
attempting to marshal recursive objects will crash your Python interpreter. Object sharing happens when there are multiple references to the same object in different places in the object hierarchy being serialized. pickle stores such objects only once, and ensures that all other references point to the master copy. Shared objects remain shared, which can be very important for mutable objects.

- **marshal** cannot be used to serialize user-defined classes and their instances. pickle can save and restore class instances transparently, however the class definition must be importable and live in the same module as when the object was stored.

- The **marshal** serialization format is not guaranteed to be portable across Python versions. Because its primary job in life is to support .pyc files, the Python implementers reserve the right to change the serialization format in non-backwards compatible ways should the need arise. The **pickle** serialization format is guaranteed to be backwards compatible across Python releases.

Note that serialization is a more primitive notion than persistence; although **pickle** reads and writes file objects, it does not handle the issue of naming persistent objects, nor the (even more complicated) issue of concurrent access to persistent objects. The **pickle** module can transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure. Perhaps the most obvious thing to do with these byte streams is to write them onto a file, but it is also conceivable to send them across a network or store them in a database. The module **shelve** provides a simple interface to pickle and unpickle objects on DBM-style database files.
11.1.2. Data stream format

The data format used by pickle is Python-specific. This has the advantage that there are no restrictions imposed by external standards such as XDR (which can’t represent pointer sharing); however it means that non-Python programs may not be able to reconstruct pickled Python objects.

By default, the pickle data format uses a printable ASCII representation. This is slightly more voluminous than a binary representation. The big advantage of using printable ASCII (and of some other characteristics of pickle's representation) is that for debugging or recovery purposes it is possible for a human to read the pickled file with a standard text editor.

There are currently 3 different protocols which can be used for pickling.

- Protocol version 0 is the original ASCII protocol and is backwards compatible with earlier versions of Python.
- Protocol version 1 is the old binary format which is also compatible with earlier versions of Python.
- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of new-style classes.

Refer to PEP 307 for more information.

If a protocol is not specified, protocol 0 is used. If protocol is specified as a negative value or HIGHEST_PROTOCOL, the highest protocol version available will be used.

Changed in version 2.3: Introduced the protocol parameter.

A binary format, which is slightly more efficient, can be chosen by
specifying a protocol version >= 1.
11.1.3. Usage

To serialize an object hierarchy, you first create a pickler, then you call the pickler’s `dump()` method. To de-serialize a data stream, you first create an unpickler, then you call the unpickler’s `load()` method.

The `pickle` module provides the following constant:

pickle. **HIGHEST_PROTOCOL**

The highest protocol version available. This value can be passed as a `protocol` value.

*New in version 2.3.*

**Note:** Be sure to always open pickle files created with protocols `>= 1` in binary mode. For the old ASCII-based pickle protocol 0 you can use either text mode or binary mode as long as you stay consistent.

A pickle file written with protocol 0 in binary mode will contain lone linefeeds as line terminators and therefore will look “funny” when viewed in Notepad or other editors which do not support this format.

The `pickle` module provides the following functions to make the pickling process more convenient:

pickle. **dump(obj, file[, protocol])**

Write a pickled representation of `obj` to the open file object `file`. This is equivalent to `Pickler(file, protocol).dump(obj).

If the `protocol` parameter is omitted, protocol 0 is used. If `protocol` is specified as a negative value or `HIGHEST_PROTOCOL`, the highest protocol version will be used.
Changed in version 2.3: Introduced the protocol parameter.

file must have a write() method that accepts a single string argument. It can thus be a file object opened for writing, a StringIO object, or any other custom object that meets this interface.

pickle. load(file)

Read a string from the open file object file and interpret it as a pickle data stream, reconstructing and returning the original object hierarchy. This is equivalent to Unpickler(file).load().

file must have two methods, a read() method that takes an integer argument, and a readline() method that requires no arguments. Both methods should return a string. Thus file can be a file object opened for reading, a StringIO object, or any other custom object that meets this interface.

This function automatically determines whether the data stream was written in binary mode or not.

pickle. dumps(obj[, protocol])

Return the pickled representation of the object as a string, instead of writing it to a file.

If the protocol parameter is omitted, protocol 0 is used. If protocol is specified as a negative value or HIGHEST_PROTOCOL, the highest protocol version will be used.

Changed in version 2.3: The protocol parameter was added.

pickle. loads(string)

Read a pickled object hierarchy from a string. Characters in the string past the pickled object's representation are ignored.
The `pickle` module also defines three exceptions:

**exception pickle.**PickleError

A common base class for the other exceptions defined below. This inherits from `Exception`.

**exception pickle.**PicklingError

This exception is raised when an unpicklable object is passed to the `dump()` method.

**exception pickle.**UnpicklingError

This exception is raised when there is a problem unpickling an object. Note that other exceptions may also be raised during unpickling, including (but not necessarily limited to) `AttributeError`, `EOFError`, `ImportError`, and `IndexError`.

The `pickle` module also exports two callables [2], `Pickler` and `Unpickler`:

**class pickle.**Pickler(file[, protocol])

This takes a file-like object to which it will write a pickle data stream.

If the `protocol` parameter is omitted, protocol 0 is used. If `protocol` is specified as a negative value or `HIGHEST_PROTOCOL`, the highest protocol version will be used.

*Changed in version 2.3:* Introduced the `protocol` parameter.

`file` must have a `write()` method that accepts a single string argument. It can thus be an open file object, a `StringIO` object, or any other custom object that meets this interface.

`Pickler` objects define one (or two) public methods:
**dump(obj)**

Write a pickled representation of obj to the open file object given in the constructor. Either the binary or ASCII format will be used, depending on the value of the *protocol* argument passed to the constructor.

**clear_memo()**

Clears the pickler’s “memo”. The memo is the data structure that remembers which objects the pickler has already seen, so that shared or recursive objects pickled by reference and not by value. This method is useful when re-using picklers.

**Note:** Prior to Python 2.3, *clear_memo()* was only available on the picklers created by *cPickle*. In the *pickle* module, picklers have an instance variable called *memo* which is a Python dictionary. So to clear the memo for a *pickle* module pickler, you could do the following:

```python
mypickler.memo.clear()
```

Code that does not need to support older versions of Python should simply use *clear_memo()*.

It is possible to make multiple calls to the *dump()* method of the same *Pickler* instance. These must then be matched to the same number of calls to the *load()* method of the corresponding *Unpickler* instance. If the same object is pickled by multiple *dump()* calls, the *load()* will all yield references to the same object. [3]

*Unpickler* objects are defined as:

```python
class pickle.Unpickler(file)
```

This takes a file-like object from which it will read a pickle data stream. This class automatically determines whether the data
stream was written in binary mode or not, so it does not need a flag as in the Pickler factory.

*file* must have two methods, a `read()` method that takes an integer argument, and a `readline()` method that requires no arguments. Both methods should return a string. Thus *file* can be a file object opened for reading, a `StringIO` object, or any other custom object that meets this interface.

*Unpickler* objects have one (or two) public methods:

**load()**
Read a pickled object representation from the open file object given in the constructor, and return the reconstituted object hierarchy specified therein.

This method automatically determines whether the data stream was written in binary mode or not.

**noload()**
This is just like `load()` except that it doesn’t actually create any objects. This is useful primarily for finding what’s called “persistent ids” that may be referenced in a pickle data stream. See section *The pickle protocol* below for more details.

**Note:** the `noload()` method is currently only available on Unpickler objects created with the `cPickle` module. `pickle` module Unpicklers do not have the `noload()` method.
11.1.4. What can be pickled and unpickled?

The following types can be pickled:

- None, True, and False
- integers, long integers, floating point numbers, complex numbers
- normal and Unicode strings
- tuples, lists, sets, and dictionaries containing only picklable objects
- functions defined at the top level of a module
- built-in functions defined at the top level of a module
- classes that are defined at the top level of a module
- instances of such classes whose __dict__ or the result of calling __getstate__() is picklable (see section The pickle protocol for details).

Attempts to pickle unpicklable objects will raise the `PicklingError` exception; when this happens, an unspecified number of bytes may have already been written to the underlying file. Trying to pickle a highly recursive data structure may exceed the maximum recursion depth, a `RuntimeError` will be raised in this case. You can carefully raise this limit with `sys.setrecursionlimit()`.

Note that functions (built-in and user-defined) are pickled by “fully qualified” name reference, not by value. This means that only the function name is pickled, along with the name of the module the function is defined in. Neither the function’s code, nor any of its function attributes are pickled. Thus the defining module must be importable in the unpickling environment, and the module must contain the named object, otherwise an exception will be raised. [4]

Similarly, classes are pickled by named reference, so the same
restrictions in the unpickling environment apply. Note that none of the class’s code or data is pickled, so in the following example the class attribute attr is not restored in the unpickling environment:

```python
class Foo:
    attr = 'a class attr'

picklestring = pickle.dumps(Foo)
```

These restrictions are why picklable functions and classes must be defined in the top level of a module.

Similarly, when class instances are pickled, their class's code and data are not pickled along with them. Only the instance data are pickled. This is done on purpose, so you can fix bugs in a class or add methods to the class and still load objects that were created with an earlier version of the class. If you plan to have long-lived objects that will see many versions of a class, it may be worthwhile to put a version number in the objects so that suitable conversions can be made by the class's __setstate__() method.
11.1.5. The pickle protocol

This section describes the “pickling protocol” that defines the interface between the pickler/unpickler and the objects that are being serialized. This protocol provides a standard way for you to define, customize, and control how your objects are serialized and de-serialized. The description in this section doesn’t cover specific customizations that you can employ to make the unpickling environment slightly safer from untrusted pickle data streams; see section Subclassing Unpicklers for more details.

11.1.5.1. Pickling and unpickling normal class instances

object.__getinitargs__()

When a pickled class instance is unpickled, its __init__() method is normally not invoked. If it is desirable that the __init__() method be called on unpickling, an old-style class can define a method __getinitargs__(), which should return a tuple containing the arguments to be passed to the class constructor (__init__() for example). The __getinitargs__() method is called at pickle time; the tuple it returns is incorporated in the pickle for the instance.

object.__getnewargs__()

New-style types can provide a __getnewargs__() method that is used for protocol 2. Implementing this method is needed if the type establishes some internal invariants when the instance is created, or if the memory allocation is affected by the values passed to the __new__() method for the type (as it is for tuples and strings). Instances of a new-style class c are created using
obj = C.__new__(C, *args)

where args is the result of calling __getnewargs__() on the original object; if there is no __getnewargs__(), an empty tuple is assumed.

object.__getstate__()  
Classes can further influence how their instances are pickled; if the class defines the method __getstate__(), it is called and the return state is pickled as the contents for the instance, instead of the contents of the instance's dictionary. If there is no __getstate__() method, the instance's __dict__ is pickled.

object.__setstate__(state)  
Upon unpickling, if the class also defines the method __setstate__(), it is called with the unpickled state. [5] If there is no __setstate__() method, the pickled state must be a dictionary and its items are assigned to the new instance's dictionary. If a class defines both __getstate__() and __setstate__(), the state object needn't be a dictionary and these methods can do what they want. [6]

Note: For new-style classes, if __getstate__() returns a false value, the __setstate__() method will not be called.

Note: At unpickling time, some methods like __getattr__(), __getattribute__(), or __setattr__() may be called upon the instance. In case those methods rely on some internal invariant being true, the type should implement either __getinitargs__() or __getnewargs__() to establish such an invariant; otherwise, neither __new__() nor __init__() will be called.
11.1.5.2. Pickling and unpickling extension types

```python
object.__reduce__()
```

When the `Pickler` encounters an object of a type it knows nothing about — such as an extension type — it looks in two places for a hint of how to pickle it. One alternative is for the object to implement a `__reduce__()` method. If provided, at pickling time `__reduce__()` will be called with no arguments, and it must return either a string or a tuple.

If a string is returned, it names a global variable whose contents are pickled as normal. The string returned by `__reduce__()` should be the object’s local name relative to its module; the pickle module searches the module namespace to determine the object’s module.

When a tuple is returned, it must be between two and five elements long. Optional elements can either be omitted, or `None` can be provided as their value. The contents of this tuple are pickled as normal and used to reconstruct the object at unpickling time. The semantics of each element are:

- A callable object that will be called to create the initial version of the object. The next element of the tuple will provide arguments for this callable, and later elements provide additional state information that will subsequently be used to fully reconstruct the pickled data.

In the unpickling environment this object must be either a class, a callable registered as a “safe constructor” (see below), or it must have an attribute `__safe_for_unpickling__` with a true value. Otherwise, an `UnpicklingError` will be raised in the unpickling environment. Note that as usual, the callable itself is pickled by name.
• A tuple of arguments for the callable object.

  *Changed in version 2.5:* Formerly, this argument could also be `None`.

• Optionally, the object's state, which will be passed to the object's `__setstate__() method as described in section Pickling and unpickling normal class instances. If the object has no `__setstate__() method, then, as above, the value must be a dictionary and it will be added to the object's `__dict__`.

• Optionally, an iterator (and not a sequence) yielding successive list items. These list items will be pickled, and appended to the object using either `obj.append(item)` or `obj.extend(list_of_items)`. This is primarily used for list subclasses, but may be used by other classes as long as they have `append()` and `extend()` methods with the appropriate signature. (Whether `append()` or `extend()` is used depends on which pickle protocol version is used as well as the number of items to append, so both must be supported.)

• Optionally, an iterator (not a sequence) yielding successive dictionary items, which should be tuples of the form `(key, value)`. These items will be pickled and stored to the object using `obj[key] = value`. This is primarily used for dictionary subclasses, but may be used by other classes as long as they implement `__setitem__()`.  

  `object.__reduce_ex__(protocol)`  
  It is sometimes useful to know the protocol version when implementing `__reduce__()`. This can be done by implementing a method named `__reduce_ex__()` instead of `__reduce__()`.  

__reduce_ex__(), when it exists, is called in preference over __reduce__() (you may still provide __reduce__() for backwards compatibility). The __reduce_ex__() method will be called with a single integer argument, the protocol version.

The object class implements both __reduce__() and __reduce_ex__(); however, if a subclass overrides __reduce__() but not __reduce_ex__(), the __reduce_ex__() implementation detects this and calls __reduce__().

An alternative to implementing a __reduce__() method on the object to be pickled, is to register the callable with the copy_reg module. This module provides a way for programs to register “reduction functions” and constructors for user-defined types. Reduction functions have the same semantics and interface as the __reduce__() method described above, except that they are called with a single argument, the object to be pickled.

The registered constructor is deemed a “safe constructor” for purposes of unpickling as described above.

11.1.5.3. Pickling and unpickling external objects

For the benefit of object persistence, the pickle module supports the notion of a reference to an object outside the pickled data stream. Such objects are referenced by a “persistent id”, which is just an arbitrary string of printable ASCII characters. The resolution of such names is not defined by the pickle module; it will delegate this resolution to user defined functions on the pickler and unpickler. [7]

To define external persistent id resolution, you need to set the persistent_id attribute of the pickler object and the persistent_load attribute of the unpickler object.
To pickle objects that have an external persistent id, the pickler must have a custom `persistent_id()` method that takes an object as an argument and returns either `None` or the persistent id for that object. When `None` is returned, the pickler simply pickles the object as normal. When a persistent id string is returned, the pickler will pickle that string, along with a marker so that the unpickler will recognize the string as a persistent id.

To unpickle external objects, the unpickler must have a custom `persistent_load()` function that takes a persistent id string and returns the referenced object.

Here's a silly example that *might* shed more light:

```python
import pickle
from cStringIO import StringIO

src = StringIO()
p = pickle.Pickler(src)

def persistent_id(obj):
    if hasattr(obj, 'x'):
        return 'the value %d' % obj.x
    else:
        return None

p.persistent_id = persistent_id

class Integer:
    def __init__(self, x):
        self.x = x
    def __str__(self):
        return 'My name is integer %d' % self.x

i = Integer(7)
print i
p.dump(i)

datastream = src.getvalue()
print repr(datastream)
dst = StringIO(datastream)
```
In the `cPickle` module, the unpickler's `persistent_load` attribute can also be set to a Python list, in which case, when the unpickler reaches a persistent id, the persistent id string will simply be appended to this list. This functionality exists so that a pickle data stream can be "sniffed" for object references without actually instantiating all the objects in a pickle. [8] Setting `persistent_load` to a list is usually used in conjunction with the `noload()` method on the Unpickler.
11.1.6. Subclassing Unpicklers

By default, unpickling will import any class that it finds in the pickle data. You can control exactly what gets unpickled and what gets called by customizing your unpickler. Unfortunately, exactly how you do this is different depending on whether you’re using pickle or cPickle. [9]

In the pickle module, you need to derive a subclass from Unpickler, overriding the load_global() method. load_global() should read two lines from the pickle data stream where the first line will the name of the module containing the class and the second line will be the name of the instance’s class. It then looks up the class, possibly importing the module and digging out the attribute, then it appends what it finds to the unpickler’s stack. Later on, this class will be assigned to the __class__ attribute of an empty class, as a way of magically creating an instance without calling its class’s __init__(). Your job (should you choose to accept it), would be to have load_global() push onto the unpickler’s stack, a known safe version of any class you deem safe to unpickle. It is up to you to produce such a class. Or you could raise an error if you want to disallow all unpickling of instances. If this sounds like a hack, you’re right. Refer to the source code to make this work.

Things are a little cleaner with cPickle, but not by much. To control what gets unpickled, you can set the unpickler’s find_global attribute to a function or None. If it is None then any attempts to unpickle instances will raise an UnpicklingError. If it is a function, then it should accept a module name and a class name, and return the corresponding class object. It is responsible for looking up the class and performing any necessary imports, and it may raise an error to prevent instances of the class from being unpickled.
The moral of the story is that you should be really careful about the source of the strings your application unpickles.
11.1.7. Example

For the simplest code, use the `dump()` and `load()` functions. Note that a self-referencing list is pickled and restored correctly.

```python
import pickle

data1 = {'a': [1, 2.0, 3, 4+6j],
         'b': ('string', u'Unicode string'),
         'c': None}

selfref_list = [1, 2, 3]
selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.
pickle.dump(data1, output)

# Pickle the list using the highest protocol available.
pickle.dump(selfref_list, output, -1)

output.close()
```

The following example reads the resulting pickled data. When reading a pickle-containing file, you should open the file in binary mode because you can’t be sure if the ASCII or binary format was used.

```python
import pprint, pickle

pkl_file = open('data.pkl', 'rb')
data1 = pickle.load(pkl_file)
pprint.pprint(data1)

data2 = pickle.load(pkl_file)
pprint.pprint(data2)
pkl_file.close()
```
Here's a larger example that shows how to modify pickling behavior for a class. The `TextReader` class opens a text file, and returns the line number and line contents each time its `readline()` method is called. If a `TextReader` instance is pickled, all attributes `except` the file object member are saved. When the instance is unpickled, the file is reopened, and reading resumes from the last location. The `__setstate__()` and `__getstate__()` methods are used to implement this behavior.

```python
#!usr/local/bin/python

class TextReader:
    """Print and number lines in a text file."""
    def __init__(self, file):
        self.file = file
        self.fh = open(file)
        self.lineno = 0

    def readline(self):
        self.lineno = self.lineno + 1
        line = self.fh.readline()
        if not line:
            return None
        if line.endswith("\n"):
            line = line[:-1]
        return "%d: %s" % (self.lineno, line)

    def __getstate__(self):
        odict = self.__dict__.copy()  # copy the dict since we c.
        del odict['fh']  # remove filehandle entry
        return odict

    def __setstate__(self, dict):
        fh = open(dict['file'])  # reopen file
        count = dict['lineno']  # read from file...
        while count:
            fh.readline()
            count = count - 1
        self.__dict__.update(dict)  # update attributes
        self.fh = fh  # save the file object
```
A sample usage might be something like this:

```python
>>> import TextReader
>>> obj = TextReader.TextReader("TextReader.py")
>>> obj.readline()
'1: #!/usr/local/bin/python'
>>> obj.readline()
'2: '
>>> obj.readline()
'3: class TextReader:
>>> import pickle
>>> pickle.dump(obj, open('save.p', 'wb'))
```

If you want to see that `pickle` works across Python processes, start another Python session, before continuing. What follows can happen from either the same process or a new process.

```python
>>> import pickle
>>> reader = pickle.load(open('save.p', 'rb'))
>>> reader.readline()
'4: """Print and number lines in a text file."""
```

See also:

**Module copy_reg**

Pickle interface constructor registration for extension types.

**Module shelve**

Indexed databases of objects; uses pickle.

**Module copy**

Shallow and deep object copying.

**Module marshal**

High-performance serialization of built-in types.
11.2. cPickle — A faster pickle

The cPickle module supports serialization and de-serialization of Python objects, providing an interface and functionality nearly identical to the pickle module. There are several differences, the most important being performance and subclassability.

First, cPickle can be up to 1000 times faster than pickle because the former is implemented in C. Second, in the cPickle module the callables Pickler() and Unpickler() are functions, not classes. This means that you cannot use them to derive custom pickling and unpickling subclasses. Most applications have no need for this functionality and should benefit from the greatly improved performance of the cPickle module.

The pickle data stream produced by pickle and cPickle are identical, so it is possible to use pickle and cPickle interchangeably with existing pickles. [10]

There are additional minor differences in API between cPickle and pickle, however for most applications, they are interchangeable. More documentation is provided in the pickle module documentation, which includes a list of the documented differences.

Footnotes

[1] Don’t confuse this with the marshal module
In the pickle module these callables are classes, which you could subclass to customize the behavior. However, in the cPickle module these callables are factory functions and so cannot be subclassed. One common reason to subclass is to control what objects can actually be unpickled. See section Subclassing Unpicklers for more details.
Warning: this is intended for pickling multiple objects without intervening modifications to the objects or their parts. If you modify an object and then pickle it again using the same
Pickler instance, the object is not pickled again — a reference to it is pickled and the Unpickler will return the old value, not the modified one. There are two problems here: (1) detecting changes, and (2) marshalling a minimal set of changes. Garbage Collection may also become a problem here.

The exception raised will likely be an ImportError or an AttributeError but it could be something else.

These methods can also be used to implement copying class instances.

This protocol is also used by the shallow and deep copying operations defined in the copy module.

The actual mechanism for associating these user defined functions is slightly different for pickle and cPickle. The description given here works the same for both implementations. Users of the pickle module could also use subclassing to effect the same results, overriding the persistent_id() and persistent_load() methods in the derived classes.

We'll leave you with the image of Guido and Jim sitting around sniffing pickles in their living rooms.

A word of caution: the mechanisms described here use internal attributes and methods, which are subject to change in future versions of Python. We intend to someday provide a common interface for controlling this behavior, which will work in either pickle or cPickle.

Since the pickle data format is actually a tiny stack-oriented programming language, and some freedom is taken in the encodings of certain objects, it is possible that the two modules produce different data streams for the same input objects. However it is guaranteed that they will always be able to read each other's data streams.
11.3. copy_reg — Register pickle support functions

Note: The copy_reg module has been renamed to copyreg in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

The copy_reg module offers a way to define functions used while pickling specific objects. The pickle, cPickle, and copy modules use those functions when pickling/copying those objects. The module provides configuration information about object constructors which are not classes. Such constructors may be factory functions or class instances.

copy_reg.constructor(object)

Declares object to be a valid constructor. If object is not callable (and hence not valid as a constructor), raises TypeError.

copy_reg.pickle(type, function[, constructor])

Declares that function should be used as a “reduction” function for objects of type type; type must not be a “classic” class object. (Classic classes are handled differently; see the documentation for the pickle module for details.) function should return either a string or a tuple containing two or three elements.

The optional constructor parameter, if provided, is a callable object which can be used to reconstruct the object when called with the tuple of arguments returned by function at pickling time. TypeError will be raised if object is a class or constructor is not callable.

See the pickle module for more details on the interface expected
of \textit{function} and \textit{constructor}.
11.3.1. Example

The example below would like to show how to register a pickle function and how it will be used:

```python
>>> import copy_reg, copy, pickle
>>> class C(object):
...     def __init__(self, a):
...         self.a = a
...
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
...
>>> copy_reg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```
A “shelf” is a persistent, dictionary-like object. The difference with “dbm” databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects — anything that the pickle module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys are ordinary strings.

`shelve.open(filename, flag='c', protocol=None, writeback=False)`

Open a persistent dictionary. The filename specified is the base filename for the underlying database. As a side-effect, an extension may be added to the filename and more than one file may be created. By default, the underlying database file is opened for reading and writing. The optional `flag` parameter has the same interpretation as the `flag` parameter of `anydbm.open()`.

By default, version 0 pickles are used to serialize values. The version of the pickle protocol can be specified with the `protocol` parameter.

*Changed in version 2.3:* The `protocol` parameter was added.

Because of Python semantics, a shelf cannot know when a mutable persistent-dictionary entry is modified. By default modified objects are written *only* when assigned to the shelf (see Example). If the optional `writeback` parameter is set to `True`, all entries accessed are also cached in memory, and written back on `sync()` and `close()`; this can make it handier to mutate mutable
entries in the persistent dictionary, but, if many entries are accessed, it can consume vast amounts of memory for the cache, and it can make the close operation very slow since all accessed entries are written back (there is no way to determine which accessed entries are mutable, nor which ones were actually mutated).

Like file objects, shelf objects should be closed explicitly to ensure that the persistent data is flushed to disk.

**Warning:** Because the `shelve` module is backed by `pickle`, it is insecure to load a shelf from an untrusted source. Like with pickle, loading a shelf can execute arbitrary code.

Shelf objects support most of the methods supported by dictionaries. This eases the transition from dictionary based scripts to those requiring persistent storage.

Note, the Python 3 transition methods (`viewkeys()`, `viewvalues()`, and `viewitems()`) are not supported.

Two additional methods are supported:

**Shelf.** `sync()`
Write back all entries in the cache if the shelf was opened with `writeback` set to `True`. Also empty the cache and synchronize the persistent dictionary on disk, if feasible. This is called automatically when the shelf is closed with `close()`.

**Shelf.** `close()`
Synchronize and close the persistent `dict` object. Operations on a closed shelf will fail with a `ValueError`.

**See also:** Persistent dictionary recipe with widely supported
storage formats and having the speed of native dictionaries.
11.4.1. Restrictions

- The choice of which database package will be used (such as `dbm`, `gdbm` or `bsddb`) depends on which interface is available. Therefore it is not safe to open the database directly using `dbm`. The database is also (unfortunately) subject to the limitations of `dbm`, if it is used — this means that (the pickled representation of) the objects stored in the database should be fairly small, and in rare cases key collisions may cause the database to refuse updates.

- The `shelve` module does not support concurrent read/write access to shelved objects. (Multiple simultaneous read accesses are safe.) When a program has a shelf open for writing, no other program should have it open for reading or writing. Unix file locking can be used to solve this, but this differs across Unix versions and requires knowledge about the database implementation used.

```python
class shelve.Shelf(dict, protocol=None, writeback=False)
```

A subclass of `UserDict.DictMixin` which stores pickled values in the `dict` object.

By default, version 0 pickles are used to serialize values. The version of the pickle protocol can be specified with the `protocol` parameter. See the `pickle` documentation for a discussion of the pickle protocols.

*Changed in version 2.3:* The `protocol` parameter was added.

If the `writeback` parameter is `True`, the object will hold a cache of all entries accessed and write them back to the `dict` at sync and close times. This allows natural operations on mutable entries, but can consume much more memory and make sync and close
take a long time.

class shelve.BsddbShelf(dict, protocol=None, writeback=False)
A subclass of Shelf which exposes first(), next(), previous(),
last() and set_location() which are available in the bsddb
module but not in other database modules. The dict object
passed to the constructor must support those methods. This is
generally accomplished by calling one of bsddb.hashopen(),
bsddb.btopen() or bsddb.rnnopen(). The optional protocol and
writeback parameters have the same interpretation as for the
Shelf class.

class shelve.DbfilenameShelf(filename, flag='c', protocol=None,
writeback=False)
A subclass of Shelf which accepts a filename instead of a dict-
like object. The underlying file will be opened using
anydbm.open(). By default, the file will be created and opened for
both read and write. The optional flag parameter has the same
interpretation as for the open() function. The optional protocol
and writeback parameters have the same interpretation as for the
Shelf class.
11.4.2. Example

To summarize the interface (key is a string, data is an arbitrary object):

```python
import shelve

d = shelve.open(filename)  # open -- file may get suffix added by
                          # library

d[key] = data              # store data at key (overwrites old data if
                          # using an existing key)
data = d[key]              # retrieve a COPY of data at key (raise KeyError
                          # such key)

del d[key]                 # delete data stored at key (raises KeyError
                          # if no such key)

flag = d.has_key(key)      # true if the key exists
klist = d.keys()           # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = range(4)         # this works as expected, but...
d['xx'].append(5)         # *this doesn't!* -- d['xx'] is STILL range

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']             # extracts the copy
temp.append(5)             # mutates the copy
d['xx'] = temp             # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just
# d['xx'].append(5) and have it work as expected, BUT it would
# consume more memory and make the d.close() operation slower.

d.close()                  # close it
```

See also:

Module **anydbm**

   Generic interface to **dbm**-style databases.

Module **bsddb**
BSD db database interface.

**Module dbhash**

Thin layer around the bsddb which provides an open() function like the other database modules.

**Module dbm**

Standard Unix database interface.

**Module dumbdbm**

Portable implementation of the dbm interface.

**Module gdbm**

GNU database interface, based on the dbm interface.

**Module pickle**

Object serialization used by shelve.

**Module cPickle**

High-performance version of pickle.
11.5. **marshal** — Internal Python object serialization

This module contains functions that can read and write Python values in a binary format. The format is specific to Python, but independent of machine architecture issues (e.g., you can write a Python value to a file on a PC, transport the file to a Sun, and read it back there). Details of the format are undocumented on purpose; it may change between Python versions (although it rarely does). [1]

This is not a general “persistence” module. For general persistence and transfer of Python objects through RPC calls, see the modules **pickle** and **shelve**. The **marshal** module exists mainly to support reading and writing the “pseudo-compiled” code for Python modules of `.pyc` files. Therefore, the Python maintainers reserve the right to modify the marshal format in backward incompatible ways should the need arise. If you’re serializing and de-serializing Python objects, use the **pickle** module instead – the performance is comparable, version independence is guaranteed, and pickle supports a substantially wider range of objects than marshal.

**Warning:** The **marshal** module is not intended to be secure against erroneous or maliciously constructed data. Never unmarshal data received from an untrusted or unauthenticated source.

Not all Python object types are supported; in general, only objects whose value is independent from a particular invocation of Python can be written and read by this module. The following types are supported: booleans, integers, long integers, floating point numbers, complex numbers, strings, Unicode objects, tuples, lists, sets, frozensets, dictionaries, and code objects, where it should be
understood that tuples, lists, sets, frozensets and dictionaries are only supported as long as the values contained therein are themselves supported; and recursive lists, sets and dictionaries should not be written (they will cause infinite loops). The singletons None, Ellipsis and StopIteration can also be marshalled and unmarshalled.

**Warning:** On machines where C's `long int` type has more than 32 bits (such as the DEC Alpha), it is possible to create plain Python integers that are longer than 32 bits. If such an integer is marshaled and read back in on a machine where C's `long int` type has only 32 bits, a Python long integer object is returned instead. While of a different type, the numeric value is the same. (This behavior is new in Python 2.2. In earlier versions, all but the least-significant 32 bits of the value were lost, and a warning message was printed.)

There are functions that read/write files as well as functions operating on strings.

The module defines these functions:

```python
marshal.dump(value, file[, version])
```

Write the value on the open file. The value must be a supported type. The file must be an open file object such as `sys.stdout` or returned by `open()` or `os.popen()`. It may not be a wrapper such as TemporaryFile on Windows. It must be opened in binary mode (`'wb'` or `'w+b'`).

If the value has (or contains an object that has) an unsupported type, a `ValueError` exception is raised — but garbage data will also be written to the file. The object will not be properly read back by `load()`.
New in version 2.4: The version argument indicates the data format that \texttt{dump} should use (see below).

\texttt{marshal.load(file)}

Read one value from the open file and return it. If no valid value is read (e.g. because the data has a different Python version's incompatible marshal format), raise \texttt{EOFError}, \texttt{ValueError} or \texttt{TypeError}. The file must be an open file object opened in binary mode (\texttt{'rb'} or \texttt{'r+b'}).

\textbf{Note:} If an object containing an unsupported type was marshalled with \texttt{dump()}, \texttt{load()} will substitute \texttt{None} for the unmarshallable type.

\texttt{marshal.dumps(value[, version])}

Return the string that would be written to a file by \texttt{dump(value, file)}. The value must be a supported type. Raise a \texttt{ValueError} exception if value has (or contains an object that has) an unsupported type.

\textit{New in version 2.4:} The version argument indicates the data format that \texttt{dumps} should use (see below).

\texttt{marshal.loads(string)}

Convert the string to a value. If no valid value is found, raise \texttt{EOFError}, \texttt{ValueError} or \texttt{TypeError}. Extra characters in the string are ignored.

In addition, the following constants are defined:

\texttt{marshal.version}

Indicates the format that the module uses. Version 0 is the historical format, version 1 (added in Python 2.4) shares interned strings and version 2 (added in Python 2.5) uses a binary format.
for floating point numbers. The current version is 2.

New in version 2.4.

Footnotes

[1] The name of this module stems from a bit of terminology used by the designers of Modula-3 (amongst others), who use the term “marshalling” for shipping of data around in a self-contained form. Strictly speaking, “to marshal” means to convert some data from internal to external form (in an RPC buffer for instance) and “unmarshalling” for the reverse process.
11.6. anydbm — Generic access to DBM-style databases

**Note:** The `anydbm` module has been renamed to `dbm` in Python 3. The `2to3` tool will automatically adapt imports when converting your sources to Python 3.

`anydbm` is a generic interface to variants of the DBM database — `dbhash` (requires `bsddb`), `gdbm`, or `dbm`. If none of these modules is installed, the slow-but-simple implementation in module `dumbdbm` will be used.

```python
anydbm.open(filename[, flag[, mode]])
```

Open the database file `filename` and return a corresponding object.

If the database file already exists, the `whichdb` module is used to determine its type and the appropriate module is used; if it does not exist, the first module listed above that can be imported is used.

The optional `flag` argument must be one of these values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r'</td>
<td>Open existing database for reading only (default)</td>
</tr>
<tr>
<td>'w'</td>
<td>Open existing database for reading and writing</td>
</tr>
<tr>
<td>'c'</td>
<td>Open database for reading and writing, creating it if it doesn’t exist</td>
</tr>
<tr>
<td>'n'</td>
<td>Always create a new, empty database, open for reading and writing</td>
</tr>
</tbody>
</table>

If not specified, the default value is 'r'.
The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal \texttt{0666} (and will be modified by the prevailing umask).

**exception** \texttt{anydbm.error}

A tuple containing the exceptions that can be raised by each of the supported modules, with a unique exception also named \texttt{anydbm.error} as the first item — the latter is used when \texttt{anydbm.error} is raised.

The object returned by \texttt{open()} supports most of the same functionality as dictionaries; keys and their corresponding values can be stored, retrieved, and deleted, and the \texttt{has_key()} and \texttt{keys()} methods are available. Keys and values must always be strings.

The following example records some hostnames and a corresponding title, and then prints out the contents of the database:

```python
import anydbm

# Open database, creating it if necessary.
db = anydbm.open('cache', 'c')

# Record some values
db['www.python.org'] = 'Python Website'
db['www.cnn.com'] = 'Cable News Network'

# Loop through contents. Other dictionary methods
# such as .keys(), .values() also work.
for k, v in db.iteritems():
    print k, '\t', v

# Storing a non-string key or value will raise an exception (most
# likely a TypeError).
db['www.yahoo.com'] = 4

# Close when done.
db.close()
```
In addition to the dictionary-like methods, `anydbm` objects provide the following method:

```python
anydbm.close()
```

Close the `anydbm` database.

See also:

- **Module `dbhash`**
  BSD `db` database interface.

- **Module `dbm`**
  Standard Unix database interface.

- **Module `dumbdbm`**
  Portable implementation of the `dbm` interface.

- **Module `gdbm`**
  GNU database interface, based on the `dbm` interface.

- **Module `shelve`**
  General object persistence built on top of the Python `dbm` interface.

- **Module `whichdb`**
  Utility module used to determine the type of an existing database.
**11.7. whichdb — Guess which DBM module created a database**

**Note:** The `whichdb` module’s only function has been put into the `dbm` module in Python 3. The `2to3` tool will automatically adapt imports when converting your sources to Python 3.

The single function in this module attempts to guess which of the several simple database modules available—`dbm`, `gdbm`, or `dbhash`—should be used to open a given file.

```
whichdb whichdb(filename)
```

Returns one of the following values: `None` if the file can’t be opened because it’s unreadable or doesn’t exist; the empty string (`''`) if the file’s format can’t be guessed; or a string containing the required module name, such as `'dbm'` or `'gdbm'`. 
11. Data Persistence
**11.8. dbm — Simple “database” interface**

Note: The dbm module has been renamed to dbm.ndbm in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

The dbm module provides an interface to the Unix “(n)dbm” library. Dbm objects behave like mappings (dictionaries), except that keys and values are always strings. Printing a dbm object doesn’t print the keys and values, and the items() and values() methods are not supported.

This module can be used with the “classic” ndbm interface, the BSD DB compatibility interface, or the GNU GDBM compatibility interface. On Unix, the configure script will attempt to locate the appropriate header file to simplify building this module.

The module defines the following:

*exception dbm.error*

Raised on dbm-specific errors, such as I/O errors. KeyError is raised for general mapping errors like specifying an incorrect key.

*dbm.library*

Name of the ndbm implementation library used.

*dbm.open(filename[, flag[, mode]])*

Open a dbm database and return a dbm object. The filename argument is the name of the database file (without the .dir or .pag extensions; note that the BSD DB implementation of the interface will append the extension .db and only create one file).
The optional *flag* argument must be one of these values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r'</td>
<td>Open existing database for reading only (default)</td>
</tr>
<tr>
<td>'w'</td>
<td>Open existing database for reading and writing</td>
</tr>
<tr>
<td>'c'</td>
<td>Open database for reading and writing, creating it if it doesn’t exist</td>
</tr>
<tr>
<td>'n'</td>
<td>Always create a new, empty database, open for reading and writing</td>
</tr>
</tbody>
</table>

The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal `0666` (and will be modified by the prevailing umask).

In addition to the dictionary-like methods, *dbm* objects provide the following method:

```python
dbm.close()
```

Close the *dbm* database.

**See also:**

**Module** anydbm

Generic interface to *dbm*-style databases.

**Module** gdbm

Similar interface to the GNU GDBM library.

**Module** whichdb

Utility module used to determine the type of an existing database.
11.9. **gdbm** — GNU’s reinterpretation of **dbm**

**Note:** The **gdbm** module has been renamed to **dbm.gnu** in Python 3. The **2to3** tool will automatically adapt imports when converting your sources to Python 3.

This module is quite similar to the **dbm** module, but uses **gdbm** instead to provide some additional functionality. Please note that the file formats created by **gdbm** and **dbm** are incompatible.

The **gdbm** module provides an interface to the GNU DBM library. **gdbm** objects behave like mappings (dictionaries), except that keys and values are always strings. Printing a **gdbm** object doesn’t print the keys and values, and the **items()** and **values()** methods are not supported.

The module defines the following constant and functions:

**exception** **gdbm.error**

Raised on **gdbm**-specific errors, such as I/O errors. **KeyError** is raised for general mapping errors like specifying an incorrect key.

**gdbm.open**(filename[, flag[, mode]])

Open a **gdbm** database and return a **gdbm** object. The **filename** argument is the name of the database file.

The optional **flag** argument can be:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r'</td>
<td>Open existing database for reading only (default)</td>
</tr>
<tr>
<td>'w'</td>
<td>Open existing database for reading and writing</td>
</tr>
</tbody>
</table>
Open database for reading and writing, creating it if it doesn’t exist

Always create a new, empty database, open for reading and writing

The following additional characters may be appended to the flag to control how the database is opened:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘c’</td>
<td>Open database in fast mode. Writes to the database will not be synchronized.</td>
</tr>
<tr>
<td>‘s’</td>
<td>Synchronized mode. This will cause changes to the database to be immediately written to the file.</td>
</tr>
<tr>
<td>‘u’</td>
<td>Do not lock database.</td>
</tr>
</tbody>
</table>

Not all flags are valid for all versions of `gdbm`. The module constant `open_flags` is a string of supported flag characters. The exception `error` is raised if an invalid flag is specified.

The optional `mode` argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0666.

In addition to the dictionary-like methods, `gdbm` objects have the following methods:

`gdbm.firstkey()`

It’s possible to loop over every key in the database using this method and the `nextkey()` method. The traversal is ordered by `gdbm`’s internal hash values, and won’t be sorted by the key values. This method returns the starting key.

`gdbm.nextkey(key)`

Returns the key that follows `key` in the traversal. The following code prints every key in the database `db`, without having to create a list in memory that contains them all:
```python
k = db.firstkey()
while k != None:
    print k
    k = db.nextkey(k)
```

gdbm. **reorganize()**

If you have carried out a lot of deletions and would like to shrink the space used by the `gdbm` file, this routine will reorganize the database. `gdbm` will not shorten the length of a database file except by using this reorganization; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

gdbm. **sync()**

When the database has been opened in fast mode, this method forces any unwritten data to be written to the disk.

gdbm. **close()**

Close the `gdbm` database.

**See also:**

**Module** anydbm

Generic interface to `dbm`-style databases.

**Module** whichdb

Utility module used to determine the type of an existing database.
11.10. `dbhash` — DBM-style interface to the BSD database library

*Deprecated since version 2.6:* The `dbhash` module has been removed in Python 3.

The `dbhash` module provides a function to open databases using the BSD `db` library. This module mirrors the interface of the other Python database modules that provide access to DBM-style databases. The `bsddb` module is required to use `dbhash`.

This module provides an exception and a function:

`exception dbhash.error`

Exception raised on database errors other than `KeyError`. It is a synonym for `bsddb.error`.

`dbhash.open(path[, flag[, mode]])`

Open a `db` database and return the database object. The `path` argument is the name of the database file.

The `flag` argument can be:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r'</td>
<td>Open existing database for reading only (default)</td>
</tr>
<tr>
<td>'w'</td>
<td>Open existing database for reading and writing</td>
</tr>
<tr>
<td>'c'</td>
<td>Open database for reading and writing, creating it if it doesn’t exist</td>
</tr>
<tr>
<td>'n'</td>
<td>Always create a new, empty database, open for reading and writing</td>
</tr>
</tbody>
</table>

For platforms on which the BSD `db` library supports locking,
'1' can be appended to indicate that locking should be used.

The optional `mode` parameter is used to indicate the Unix permission bits that should be set if a new database must be created; this will be masked by the current umask value for the process.

See also:

**Module anydbm**
Generic interface to `dbm`-style databases.

**Module bsddb**
Lower-level interface to the BSD `db` library.

**Module whichdb**
Utility module used to determine the type of an existing database.
11.10.1. Database Objects

The database objects returned by `open()` provide the methods common to all the DBM-style databases and mapping objects. The following methods are available in addition to the standard methods.

`dbh.hash.first()`

It's possible to loop over every key/value pair in the database using this method and the `next()` method. The traversal is ordered by the databases internal hash values, and won't be sorted by the key values. This method returns the starting key.

`dbh.hash.last()`

Return the last key/value pair in a database traversal. This may be used to begin a reverse-order traversal; see `previous()`.

`dbh.hash.next()`

Returns the key next key/value pair in a database traversal. The following code prints every key in the database `db`, without having to create a list in memory that contains them all:

```python
print db.first()
for i in xrange(1, len(db)):
    print db.next()
```

`dbh.hash.previous()`

Returns the previous key/value pair in a forward-traversal of the database. In conjunction with `last()`, this may be used to implement a reverse-order traversal.

`dbh.hash.sync()`

This method forces any unwritten data to be written to the disk.
11.11. bsddb — Interface to Berkeley DB library

 Deprecated since version 2.6: The bsddb module has been removed in Python 3.

The bsddb module provides an interface to the Berkeley DB library. Users can create hash, btree or record based library files using the appropriate open call. Bsddb objects behave generally like dictionaries. Keys and values must be strings, however, so to use other objects as keys or to store other kinds of objects the user must serialize them somehow, typically using marshal.dumps() or pickle.dumps().

The bsddb module requires a Berkeley DB library version from 4.0 thru 4.7.

See also:

http://www.jcea.es/programacion/pybsddb.htm
  The website with documentation for the bsddb.db Python Berkeley DB interface that closely mirrors the object oriented interface provided in Berkeley DB 4.x itself.

http://www.oracle.com/database/berkeley-db/
  The Berkeley DB library.

A more modern DB, DBEnv and DBSequence object interface is available in the bsddb.db module which closely matches the Berkeley DB C API documented at the above URLs. Additional features provided by the bsddb.db API include fine tuning, transactions,
logging, and multiprocess concurrent database access.

The following is a description of the legacy `bsddb` interface compatible with the old Python bsddb module. Starting in Python 2.5 this interface should be safe for multithreaded access. The `bsddb.db` API is recommended for threading users as it provides better control.

The `bsddb` module defines the following functions that create objects that access the appropriate type of Berkeley DB file. The first two arguments of each function are the same. For ease of portability, only the first two arguments should be used in most instances.

```python
bsddb.hashopen(filename[, flag[, mode[, pgsiz[, ffactor[, nelem[, cachesize[, lorder[, hflags]]]]]]]])
```

Open the hash format file named `filename`. Files never intended to be preserved on disk may be created by passing `None` as the `filename`. The optional `flag` identifies the mode used to open the file. It may be `'r'` (read only), `'w'` (read-write), `'c'` (read-write - create if necessary; the default) or `'n'` (read-write - truncate to zero length). The other arguments are rarely used and are just passed to the low-level `dbopen()` function. Consult the Berkeley DB documentation for their use and interpretation.

```python
bsddb.btopen(filename[, flag[, mode[, btflags[, cachesize[, maxkeypage[, minkeypage[, pgsiz[, lorder]]]]]]]])
```

Open the btree format file named `filename`. Files never intended to be preserved on disk may be created by passing `None` as the `filename`. The optional `flag` identifies the mode used to open the file. It may be `'r'` (read only), `'w'` (read-write), `'c'` (read-write - create if necessary; the default) or `'n'` (read-write - truncate to zero length). The other arguments are rarely used and are just passed to the low-level `dbopen()` function. Consult the Berkeley DB documentation for their use and interpretation.
`bsddb`.

```python
def ropen(filename[, flag[, mode[, rnflags[, cachesize[, pgsize[, lorder[, rlen[, delim[, source[, pad]]]]]]]]]]):
```

Open a DB record format file named `filename`. Files never intended to be preserved on disk may be created by passing `None` as the `filename`. The optional `flag` identifies the mode used to open the file. It may be `'r'` (read only), `'w'` (read-write), `'c'` (read-write - create if necessary; the default) or `'n'` (read-write - truncate to zero length). The other arguments are rarely used and are just passed to the low-level `dbopen` function. Consult the Berkeley DB documentation for their use and interpretation.

**Note:** Beginning in 2.3 some Unix versions of Python may have a `bsddb185` module. This is present *only* to allow backwards compatibility with systems which ship with the old Berkeley DB 1.85 database library. The `bsddb185` module should never be used directly in new code. The module has been removed in Python 3. If you find you still need it look in PyPI.

**See also:**

**Module** `dbhash`

DBM-style interface to the `bsddb`
11.11.1. Hash, BTree and Record Objects

Once instantiated, hash, btrees and record objects support the same methods as dictionaries. In addition, they support the methods listed below.

*Changed in version 2.3.1:* Added dictionary methods.

**bsddbobject. close()**

Close the underlying file. The object can no longer be accessed. Since there is no open open() method for these objects, to open the file again a new bsddb module open function must be called.

**bsddbobject. keys()**

Return the list of keys contained in the DB file. The order of the list is unspecified and should not be relied on. In particular, the order of the list returned is different for different file formats.

**bsddbobject. has_key(key)**

Return 1 if the DB file contains the argument as a key.

**bsddbobject. set_location(key)**

Set the cursor to the item indicated by key and return a tuple containing the key and its value. For binary tree databases (opened using btope(), if key does not actually exist in the database, the cursor will point to the next item in sorted order and return that key and value. For other databases, KeyError will be raised if key is not found in the database.

**bsddbobject. first()**

Set the cursor to the first item in the DB file and return it. The order of keys in the file is unspecified, except in the case of B-Tree databases. This method raises bsddb.error if the database...
is empty.

bsddbobject.next()
Set the cursor to the next item in the DB file and return it. The order of keys in the file is unspecified, except in the case of B-Tree databases.

bsddbobject.previous()
Set the cursor to the previous item in the DB file and return it. The order of keys in the file is unspecified, except in the case of B-Tree databases. This is not supported on hashtable databases (those opened with hashopen()).

bsddbobject.last()
Set the cursor to the last item in the DB file and return it. The order of keys in the file is unspecified. This is not supported on hashtable databases (those opened with hashopen()). This method raises bsddb.error if the database is empty.

bsddbobject.sync()
Synchronize the database on disk.

Example:

```python
>>> import bsddb
>>> db = bsddb.btopen('spam.db', 'c')
>>> for i in range(10): db['%d' %i] = '%d' % (i*i)
... >>> db['3']
'9'
>>> db.keys()
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> db.first()
('0', '0')
>>> db.next()
('1', '1')
>>> db.last()
('9', '81')
>>> db.set_location('2')
```
('2', '4')
>>> db.previous()
('1', '1')
>>> for k, v in db.iteritems():
...     print k, v
0 0
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81

>>> '8' in db
True
>>> db.sync()
0
11.12. dumbdbm — Portable DBM implementation

**Note:** The `dumbdbm` module has been renamed to `dbm.dumb` in Python 3. The `2to3` tool will automatically adapt imports when converting your sources to Python 3.

**Note:** The `dumbdbm` module is intended as a last resort fallback for the `anydbm` module when no more robust module is available. The `dumbdbm` module is not written for speed and is not nearly as heavily used as the other database modules.

The `dumbdbm` module provides a persistent dictionary-like interface which is written entirely in Python. Unlike other modules such as `gdbm` and `bsddb`, no external library is required. As with other persistent mappings, the keys and values must always be strings.

The module defines the following:

**exception** `dumbdbm.error`

Raised on dumbdbm-specific errors, such as I/O errors. `KeyError` is raised for general mapping errors like specifying an incorrect key.

`dumbdbm.open(filename[, flag[, mode]])`

Open a dumbdbm database and return a dumbdbm object. The `filename` argument is the basename of the database file (without any specific extensions). When a dumbdbm database is created, files with `.dat` and `.dir` extensions are created.

The optional `flag` argument is currently ignored; the database is always opened for update, and will be created if it does not exist.
The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0666 (and will be modified by the prevailing umask).

*Changed in version 2.2:* The *mode* argument was ignored in earlier versions.

In addition to the dictionary-like methods, `dumbdbm` objects provide the following method:

```
dumbdbm.close()
```

Close the `dumbdbm` database.

---

**See also:**

- **Module** `anydbm`
  
  Generic interface to `dbm`-style databases.

- **Module** `dbm`
  
  Similar interface to the DBM/NDBM library.

- **Module** `gdbm`
  
  Similar interface to the GNU GDBM library.

- **Module** `shelve`
  
  Persistence module which stores non-string data.

- **Module** `whichdb`
  
  Utility module used to determine the type of an existing database.
11.12.1. Dumbdbm Objects

In addition to the methods provided by the UserDict.DictMixin class, dumbdbm objects provide the following methods.

```
dumbdbm.sync()
```

Synchronize the on-disk directory and data files. This method is called by the sync() method of shelfe objects.
11.13. sqlite3 — DB-API 2.0 interface for SQLite databases

New in version 2.5.

SQLite is a C library that provides a lightweight disk-based database that doesn’t require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It’s also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

The sqlite3 module was written by Gerhard Häring. It provides a SQL interface compliant with the DB-API 2.0 specification described by PEP 249.

To use the module, you must first create a Connection object that represents the database. Here the data will be stored in the example.db file:

```python
import sqlite3
conn = sqlite3.connect('example.db')
```

You can also supply the special name :memory: to create a database in RAM.

Once you have a Connection, you can create a Cursor object and call its execute() method to perform SQL commands:

```python
c = conn.cursor()
#
# Create table
# Create table
```
# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)"

# Save (commit) the changes
c.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
c.close()

The data you’ve saved is persistent and is available in subsequent sessions:

```python
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()

```

Usually your SQL operations will need to use values from Python variables. You shouldn’t assemble your query using Python’s string operations because doing so is insecure; it makes your program vulnerable to an SQL injection attack (see https://xkcd.com/327/ for humorous example of what can go wrong).

Instead, use the DB-API’s parameter substitution. Put `?` as a placeholder wherever you want to use a value, and then provide a tuple of values as the second argument to the cursor’s `execute()` method. (Other database modules may use a different placeholder, such as `%s` or `:1`.) For example:

```python
# Never do this -- insecure!
symbol = 'RHAT'
c.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)

# Do this instead
t = ('RHAT',)
c.execute('SELECT * FROM stocks WHERE symbol=?', t)
print c.fetchone()

# Larger example that inserts many records at a time
purchases = [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
             ('2006-04-05', 'BUY', 'MSFT', 1000, 72.00),
             ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
             ]
c.executemany('INSERT INTO stocks VALUES (?, ?, ?, ?, ?)', purchases)

To retrieve data after executing a SELECT statement, you can either
treat the cursor as an iterator, call the cursor's `fetchone()` method to
retrieve a single matching row, or call `fetchall()` to get a list of the
matching rows.

This example uses the iterator form:

```python
>>> for row in c.execute('SELECT * FROM stocks ORDER BY price'):
    print row
(u'2006-01-05', u'BUY', u'RHAT', 100, 35.14)
(u'2006-03-28', u'BUY', u'IBM', 1000, 45.0)
(u'2006-04-06', u'SELL', u'IBM', 500, 53.0)
(u'2006-04-05', u'BUY', u'MSFT', 1000, 72.0)
```

See also:

[https://github.com/ghaering/pysqlite](https://github.com/ghaering/pysqlite)

The pysqlite web page – sqlite3 is developed externally under
the name “pysqlite”.

[https://www.sqlite.org](https://www.sqlite.org)

The SQLite web page; the documentation describes the syntax
and the available data types for the supported SQL dialect.

[http://www.w3schools.com/sql/](http://www.w3schools.com/sql/)

Tutorial, reference and examples for learning SQL syntax.

**PEP 249 - Database API Specification 2.0**

PEP written by Marc-André Lemburg.
11.13.1. Module functions and constants

**sqlite3.version**

The version number of this module, as a string. This is not the version of the SQLite library.

**sqlite3.version_info**

The version number of this module, as a tuple of integers. This is not the version of the SQLite library.

**sqlite3.sqlite_version**

The version number of the run-time SQLite library, as a string.

**sqlite3.sqlite_version_info**

The version number of the run-time SQLite library, as a tuple of integers.

**sqlite3.PARSE_DECLTYPES**

This constant is meant to be used with the *detect_types* parameter of the *connect()* function.

Setting it makes the sqlite3 module parse the declared type for each column it returns. It will parse out the first word of the declared type, i.e. for “integer primary key”, it will parse out “integer”, or for “number(10)” it will parse out “number”. Then for that column, it will look into the converters dictionary and use the converter function registered for that type there.

**sqlite3.PARSE_COLNAMES**

This constant is meant to be used with the *detect_types* parameter of the *connect()* function.

Setting this makes the SQLite interface parse the column name for each column it returns. It will look for a string formed [mytype] in there, and then decide that ‘mytype’ is the type of the column.
It will try to find an entry of ‘mytype’ in the converters dictionary and then use the converter function found there to return the value. The column name found in `Cursor.description` is only the first word of the column name, i.e. if you use something like 'as "x [datetime]"' in your SQL, then we will parse out everything until the first blank for the column name: the column name would simply be “x”.

```python
sqlite3.connect(database[, timeout, detect_types, isolation_level,
check_same_thread, factory, cached_statements])
```

Opens a connection to the SQLite database file `database`. You can use `":memory:"` to open a database connection to a database that resides in RAM instead of on disk.

When a database is accessed by multiple connections, and one of the processes modifies the database, the SQLite database is locked until that transaction is committed. The `timeout` parameter specifies how long the connection should wait for the lock to go away until raising an exception. The default for the timeout parameter is 5.0 (five seconds).

For the `isolation_level` parameter, please see the `Connection.isolation_level` property of `Connection` objects.

SQLite natively supports only the types TEXT, INTEGER, REAL, BLOB and NULL. If you want to use other types you must add support for them yourself. The `detect_types` parameter and the using custom `converters` registered with the module-level `register_converter()` function allow you to easily do that.

`detect_types` defaults to 0 (i.e. off, no type detection), you can set it to any combination of `PARSE_DECLTYPES` and `PARSE_COLNAMES` to turn type detection on.
By default, the `sqlite3` module uses its `Connection` class for the connect call. You can, however, subclass the `Connection` class and make `connect()` use your class instead by providing your class for the `factory` parameter.

Consult the section SQLite and Python types of this manual for details.

The `sqlite3` module internally uses a statement cache to avoid SQL parsing overhead. If you want to explicitly set the number of statements that are cached for the connection, you can set the `cached_statements` parameter. The currently implemented default is to cache 100 statements.

`sqlite3.register_converter(typename, callable)`

Registers a callable to convert a bytestring from the database into a custom Python type. The callable will be invoked for all database values that are of the type `typename`. Confer the parameter `detect_types` of the `connect()` function for how the type detection works. Note that the case of `typename` and the name of the type in your query must match!

`sqlite3.register_adapter(type, callable)`

Registers a callable to convert the custom Python type `type` into one of SQLite's supported types. The callable `callable` accepts as single parameter the Python value, and must return a value of the following types: int, long, float, str (UTF-8 encoded), unicode or buffer.

`sqlite3.complete_statement(sql)`

Returns `True` if the string `sql` contains one or more complete SQL statements terminated by semicolons. It does not verify that the SQL is syntactically correct, only that there are no unclosed string literals and the statement is terminated by a semicolon.
This can be used to build a shell for SQLite, as in the following example:

```python
# A minimal SQLite shell for experiments
import sqlite3
con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()

buffer = ""

print "Enter your SQL commands to execute in sqlite3."
print "Enter a blank line to exit."

while True:
    line = raw_input()
    if line == "":
        break
    buffer += line
    if sqlite3.complete_statement(buffer):
        try:
            buffer = buffer.strip()
            cur.execute(buffer)

            if buffer.lstrip().upper().startswith("SELECT"):
                print cur.fetchall()
        except sqlite3.Error as e:
            print "An error occurred:", e.args[0]
            buffer = ""

con.close()
```

`sqlite3.enable_callback_tracebacks(flag)`

By default you will not get any tracebacks in user-defined functions, aggregates, converters, authorizer callbacks etc. If you want to debug them, you can call this function with `flag` set to `True`. Afterwards, you will get tracebacks from callbacks on `sys.stderr`. Use `False` to disable the feature again.
class sqlite3.Connection

A SQLite database connection has the following attributes and methods:

isolation_level
Get or set the current isolation level. `None` for autocommit mode or one of “DEFERRED”, “IMMEDIATE” or “EXCLUSIVE”. See section Controlling Transactions for a more detailed explanation.

cursor(factory=Cursor)
The cursor method accepts a single optional parameter `factory`. If supplied, this must be a callable returning an instance of `Cursor` or its subclasses.

commit()
This method commits the current transaction. If you don’t call this method, anything you did since the last call to `commit()` is not visible from other database connections. If you wonder why you don’t see the data you’ve written to the database, please check you didn’t forget to call this method.

rollback()
This method rolls back any changes to the database since the last call to `commit()`.

close()
This closes the database connection. Note that this does not automatically call `commit()`. If you just close your database connection without calling `commit()` first, your changes will be lost!
**execute**(sql[, parameters])

This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor’s `execute` method with the parameters given.

**executemany**(sql[, parameters])

This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor’s `executemany` method with the parameters given.

**executescript**(sql_script)

This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor’s `executescript` method with the parameters given.

**create_function**(name, num_params, func)

Creates a user-defined function that you can later use from within SQL statements under the function name `name`. `num_params` is the number of parameters the function accepts, and `func` is a Python callable that is called as the SQL function.

The function can return any of the types supported by SQLite: `unicode`, `str`, `int`, `long`, `float`, `buffer` and `None`.

Example:

```python
import sqlite3
import md5

def md5sum(t):
    return md5.md5(t).hexdigest()

con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", ("foo",))
```
create_aggregate(name, num_params, aggregate_class)

Creates a user-defined aggregate function.

The aggregate class must implement a step method, which accepts the number of parameters num_params, and a finalize method which will return the final result of the aggregate.

The finalize method can return any of the types supported by SQLite: unicode, str, int, long, float, buffer and None.

Example:

```python
import sqlite3
class MySum:
    def __init__(self):
        self.count = 0
    def step(self, value):
        self.count += value
    def finalize(self):
        return self.count
con = sqlite3.connect(':memory:)
con.create_aggregate("mysum", 1, MySum)
cur = con.cursor()
cur.execute("create table test(i)")
cur.execute("insert into test(i) values (1)"
cur.execute("insert into test(i) values (2)"
cur.execute("select mysum(i) from test")
print cur.fetchone()[0]
```

create_collation(name, callable)

Creates a collation with the specified name and callable. The callable will be passed two string arguments. It should return -1 if the first is ordered lower than the second, 0 if they are
ordered equal and 1 if the first is ordered higher than the second. Note that this controls sorting (ORDER BY in SQL) so your comparisons don’t affect other SQL operations.

Note that the callable will get its parameters as Python bytestrings, which will normally be encoded in UTF-8.

The following example shows a custom collation that sorts “the wrong way”:

```python
import sqlite3

def collate_reverse(string1, string2):
    return -cmp(string1, string2)

con = sqlite3.connect(':memory:)
con.create_collation('reverse', collate_reverse)

cur = con.cursor()
cur.execute("create table test(x)")
cur.executemany("insert into test(x) values (?)", [("a",)])
cur.execute("select x from test order by x collate reverse")
for row in cur:
    print row
con.close()
```

To remove a collation, call `create_collation` with `None` as callable:

```python
con.create_collation('reverse', None)
```

`interrupt()`

You can call this method from a different thread to abort any queries that might be executing on the connection. The query will then abort and the caller will get an exception.

`set_authorizer(authorizer_callback)`

This routine registers a callback. The callback is invoked for
each attempt to access a column of a table in the database. The callback should return 
\texttt{SQLITE\_OK} if access is allowed, \texttt{SQLITE\_DENY} if the entire SQL statement should be aborted with an error and \texttt{SQLITE\_IGNORE} if the column should be treated as a NULL value. These constants are available in the \texttt{sqlite3} module.

The first argument to the callback signifies what kind of operation is to be authorized. The second and third argument will be arguments or \texttt{None} depending on the first argument. The 4th argument is the name of the database ("main", "temp", etc.) if applicable. The 5th argument is the name of the inner-most trigger or view that is responsible for the access attempt or \texttt{None} if this access attempt is directly from input SQL code.

Please consult the SQLite documentation about the possible values for the first argument and the meaning of the second and third argument depending on the first one. All necessary constants are available in the \texttt{sqlite3} module.

\textbf{set\_progress\_handler}(\texttt{handler, n})

This routine registers a callback. The callback is invoked for every \textit{n} instructions of the SQLite virtual machine. This is useful if you want to get called from SQLite during long-running operations, for example to update a GUI.

If you want to clear any previously installed progress handler, call the method with \texttt{None} for \texttt{handler}.

\textit{New in version 2.6.}

\textbf{enable\_load\_extension}(\texttt{enabled})

This routine allows/disallows the SQLite engine to load SQLite
extensions from shared libraries. SQLite extensions can define new functions, aggregates or whole new virtual table implementations. One well-known extension is the fulltext-search extension distributed with SQLite.

Loadable extensions are disabled by default. See [1].

New in version 2.7.

```python
import sqlite3
con = sqlite3.connect(":memory:"

# enable extension loading
con.enable_load_extension(True)

# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using an API call:
# con.load_extension('./fts3.so')

# disable extension loading again
con.enable_load_extension(False)

# example from SQLite wiki
con.execute("create virtual table recipe using fts3(name, ingredients)"
con.executescript(""
    insert into recipe (name, ingredients) values ('broccoli stew', 'broccoli peppers cheese tomatoes');
    insert into recipe (name, ingredients) values ('pumpkin stew', 'pumpkin onions garlic celery');
    insert into recipe (name, ingredients) values ('broccoli pie', 'broccoli cheese onions flour');
    insert into recipe (name, ingredients) values ('pumpkin pie', 'pumpkin sugar flour butter');
"")

for row in con.execute("select rowid, name, ingredients from recipe where name match 'pie'
print row
```

`load_extension(path)`

This routine loads a SQLite extension from a shared library. You have to enable extension loading with `enable_load_extension()` before you can use this routine.
Loadable extensions are disabled by default. See [1].

*New in version 2.7.*

**row_factory**

You can change this attribute to a callable that accepts the cursor and the original row as a tuple and will return the real result row. This way, you can implement more advanced ways of returning results, such as returning an object that can also access columns by name.

Example:

```python
import sqlite3

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d

con = sqlite3.connect(':memory:)
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print cur.fetchone()["a"]
```

If returning a tuple doesn’t suffice and you want name-based access to columns, you should consider setting `row_factory` to the highly-optimized `sqlite3.Row` type. `Row` provides both index-based and case-insensitive name-based access to columns with almost no memory overhead. It will probably be better than your own custom dictionary-based approach or even a db_row based solution.

**text_factory**

Using this attribute you can control what objects are returned for the `TEXT` data type. By default, this attribute is set to
unicode and the sqlite3 module will return Unicode objects for `TEXT`. If you want to return bytestrings instead, you can set it to `str`.

For efficiency reasons, there's also a way to return Unicode objects only for non-ASCII data, and bytestrings otherwise. To activate it, set this attribute to `sqlite3.OptimizedUnicode`.

You can also set it to any other callable that accepts a single bytestring parameter and returns the resulting object.

See the following example code for illustration:

```python
import sqlite3

con = sqlite3.connect(':memory:)
cur = con.cursor()

AUSTRIA = u'Österreich'

# by default, rows are returned as Unicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert row[0] == AUSTRIA

# but we can make sqlite3 always return bytestrings ...
con.text_factory = str
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is str
# the bytestrings will be encoded in UTF-8, unless you store garbage in the database ...
assert row[0] == AUSTRIA.encode("utf-8")

# we can also implement a custom text_factory ...
# here we implement one that will ignore Unicode characters that cannot be decoded from UTF-8
con.text_factory = lambda x: unicode(x, "utf-8", "ignore")
cur.execute("select ?", ("this is latin1 and would normally print u"\xe4\xf6\xfc",_thumb))
row = cur.fetchone()
assert type(row[0]) is unicode
```
# sqlite3 offers a built-in optimized text_factor that will return bytestring objects, if the data is in ASCII only, and otherwise return unicode objects.

c.on.text_factor = sqlite3.OptimizedUnicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is unicode

cur.execute("select ?", ("Germany",))
row = cur.fetchone()
assert type(row[0]) is str

**total_changes**

Returns the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.

**iterdump**

Returns an iterator to dump the database in an SQL text format. Useful when saving an in-memory database for later restoration. This function provides the same capabilities as the `.dump` command in the `sqlite3` shell.

*New in version 2.6.*

Example:

```python
# Convert file existing_db.db to SQL dump file dump.sql

import sqlite3, os

con = sqlite3.connect('existing_db.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s
' % line)
```
class sqlite3.Cursor

A Cursor instance has the following attributes and methods.

**execute(sql[, parameters])**

Executes an SQL statement. The SQL statement may be parameterized (i.e. placeholders instead of SQL literals). The sqlite3 module supports two kinds of placeholders: question marks (qmark style) and named placeholders (named style).

Here's an example of both styles:

```python
import sqlite3

con = sqlite3.connect(':memory:)
cur = con.cursor()
cur.execute("create table people (name_last, age)"

who = "Yeltsin"
age = 72

# This is the qmark style:
cur.execute("insert into people values (?, ?)", (who, age)

# And this is the named style:
cur.execute("select * from people where name_last=:who and age=:age"

print cur.fetchone()
```

**execute()** will only execute a single SQL statement. If you try to execute more than one statement with it, it will raise a Warning. Use **executescript()** if you want to execute multiple SQL statements with one call.

**executemany(sql, seq_of_parameters)**
Executes an SQL command against all parameter sequences or mappings found in the sequence sql. The `sqlite3` module also allows using an iterator yielding parameters instead of a sequence.

```python
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
        return self

    def next(self):
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),)  # this is a 1-tuple

con = sqlite3.connect(':memory:)
cur = con.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)
cur.execute("select c from characters")
print cur.fetchall()
```

Here's a shorter example using a generator:

```python
import sqlite3
import string

def char_generator():
    for c in string.lowercase:
        yield (c)

con = sqlite3.connect(':memory:)
cur = con.cursor()
cur.execute("create table characters(c)")
```
`executemany`:

```python
cur.executemany("insert into characters(c) values (?)", char_generator)
```

`execute`:

```python
cur.execute("select c from characters")
```

`print`:

```python
print cur.fetchall()
```

`executescript` *(sql_script)*

This is a nonstandard convenience method for executing multiple SQL statements at once. It issues a `COMMIT` statement first, then executes the SQL script it gets as a parameter.

The parameter *sql_script* can be a bytestring or a Unicode string.

Example:

```python
import sqlite3
con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript(""
    create table person(
        firstname,
        lastname,
        age
    );

    create table book(
        title,
        author,
        published
    );

    insert into book(title, author, published)
    values (
        'Dirk Gently''s Holistic Detective Agency',
        'Douglas Adams',
        1987
    );
    ""
"
```

`fetchone`:

Fetches the next row of a query result set, returning a single
sequence, or None when no more data is available.

fetchmany([size=cursor.arraysize])
Fetches the next set of rows of a query result, returning a list. An empty list is returned when no more rows are available.

The number of rows to fetch per call is specified by the size parameter. If it is not given, the cursor’s arraysize determines the number of rows to be fetched. The method should try to fetch as many rows as indicated by the size parameter. If this is not possible due to the specified number of rows not being available, fewer rows may be returned.

Note there are performance considerations involved with the size parameter. For optimal performance, it is usually best to use the arraysize attribute. If the size parameter is used, then it is best for it to retain the same value from one fetchmany() call to the next.

fetchall()
Fetches all (remaining) rows of a query result, returning a list. Note that the cursor’s arraysize attribute can affect the performance of this operation. An empty list is returned when no rows are available.

rowcount
Although the Cursor class of the sqlite3 module implements this attribute, the database engine’s own support for the determination of “rows affected”/“rows selected” is quirky.

For executemany() statements, the number of modifications are summed up into rowcount.

As required by the Python DB API Spec, the rowcount attribute “is -1 in case no executeXX() has been performed on
the cursor or the rowcount of the last operation is not determinable by the interface”. This includes `SELECT` statements because we cannot determine the number of rows a query produced until all rows were fetched.

With SQLite versions before 3.6.5, `rowcount` is set to 0 if you make a `DELETE FROM` table without any condition.

`lastrowid`

This read-only attribute provides the rowid of the last modified row. It is only set if you issued an `INSERT` statement using the `execute()` method. For operations other than `INSERT` or when `executemany()` is called, `lastrowid` is set to `None`.

`description`

This read-only attribute provides the column names of the last query. To remain compatible with the Python DB API, it returns a 7-tuple for each column where the last six items of each tuple are `None`.

It is set for `SELECT` statements without any matching rows as well.

`connection`

This read-only attribute provides the SQLite database `Connection` used by the `Cursor` object. A `Cursor` object created by calling `con.cursor()` will have a `connection` attribute that refers to `con`:

```python
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
```
11.13.4. Row Objects

```
class sqlite3.Row

A Row instance serves as a highly optimized row_factory for Connection objects. It tries to mimic a tuple in most of its features.

It supports mapping access by column name and index, iteration, representation, equality testing and len().

If two Row objects have exactly the same columns and their members are equal, they compare equal.

Changed in version 2.6: Added iteration and equality (hashability).

keys()

This method returns a list of column names. Immediately after a query, it is the first member of each tuple in Cursor.description.

New in version 2.6.
```

Let’s assume we initialize a table as in the example given above:

```python
c conn = sqlite3.connect(":memory:")
c c = conn.cursor()
c c.execute('''create table stocks
        (date text, trans text, symbol text,
        qty real, price real)''')
c c.execute("""insert into stocks
        values ('2006-01-05','BUY','RHAT',100,35.14)""")
c conn.commit()
c c.close()
```

Now we plug Row in:
>>> conn.row_factory = sqlite3.Row
>>> c = conn.cursor()
>>> c.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = c.fetchone()
>>> type(r)
<type 'sqlite3.Row'>
>>> r
(u'2006-01-05', u'BUY', u'RHAT', 100.0, 35.14)
>>> len(r)
5
>>> r[2]
u'RHAT'
>>> r.keys()
['date', 'trans', 'symbol', 'qty', 'price']
>>> r['qty']
100.0
>>> for member in r:
...     print member
... 2006-01-05
BUY
RHAT
100.0
35.14
11.13.5. SQLite and Python types

11.13.5.1. Introduction

SQLite natively supports the following types: NULL, INTEGER, REAL, TEXT, BLOB.

The following Python types can thus be sent to SQLite without any problem:

<table>
<thead>
<tr>
<th>Python type</th>
<th>SQLite type</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>NULL</td>
</tr>
<tr>
<td>int</td>
<td>INTEGER</td>
</tr>
<tr>
<td>long</td>
<td>INTEGER</td>
</tr>
<tr>
<td>float</td>
<td>REAL</td>
</tr>
<tr>
<td>str (UTF8-encoded)</td>
<td>TEXT</td>
</tr>
<tr>
<td>unicode</td>
<td>TEXT</td>
</tr>
<tr>
<td>buffer</td>
<td>BLOB</td>
</tr>
</tbody>
</table>

This is how SQLite types are converted to Python types by default:

<table>
<thead>
<tr>
<th>SQLite type</th>
<th>Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>None</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int or long, depending on size</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
</tr>
<tr>
<td>TEXT</td>
<td>depends on text_factory, unicode by default</td>
</tr>
<tr>
<td>BLOB</td>
<td>buffer</td>
</tr>
</tbody>
</table>

The type system of the sqlite3 module is extensible in two ways: you can store additional Python types in a SQLite database via object adaptation, and you can let the sqlite3 module convert SQLite types to different Python types via converters.
11.13.5.2. Using adapters to store additional Python types in SQLite databases

As described before, SQLite supports only a limited set of types natively. To use other Python types with SQLite, you must adapt them to one of the sqlite3 module’s supported types for SQLite: one of NoneType, int, long, float, str, unicode, buffer.

There are two ways to enable the sqlite3 module to adapt a custom Python type to one of the supported ones.

11.13.5.2.1. Letting your object adapt itself

This is a good approach if you write the class yourself. Let’s suppose you have a class like this:

```python
class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y
```

Now you want to store the point in a single SQLite column. First you’ll have to choose one of the supported types first to be used for representing the point. Let’s just use str and separate the coordinates using a semicolon. Then you need to give your class a method `__conform__(self, protocol)` which must return the converted value. The parameter `protocol` will be `PrepareProtocol`.

```python
import sqlite3
class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "%f;%f" % (self.x, self.y)
```
con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print cur.fetchone()[0]

### 11.13.5.2.2. Registering an adapter callable

The other possibility is to create a function that converts the type to the string representation and register the function with `register_adapter()`.

**Note:** The type/class to adapt must be a [new-style class](https://docs.python.org/3/glossary.html#term-new-style-class), i.e. it must have `object` as one of its bases.

```python
import sqlite3
class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print cur.fetchone()[0]
```

The `sqlite3` module has two default adapters for Python’s built-in `datetime.date` and `datetime.datetime` types. Now let’s suppose we want to store `datetime.datetime` objects not in ISO representation, but as a Unix timestamp.

```python
import sqlite3
```
import datetime, time

def adapt_datetime(ts):
    return time.mktime(ts.timetuple())

sqlite3.register_adapter(datetime.datetime, adapt_datetime)

con = sqlite3.connect(':memory:)
cur = con.cursor()

now = datetime.datetime.now()
cur.execute("select ?", (now,))
print cur.fetchone()[0]

11.13.5.3. Converting SQLite values to custom Python types

Writing an adapter lets you send custom Python types to SQLite. But to make it really useful we need to make the Python to SQLite to Python roundtrip work.

Enter converters.

Let’s go back to the Point class. We stored the x and y coordinates separated via semicolons as strings in SQLite.

First, we’ll define a converter function that accepts the string as a parameter and constructs a Point object from it.

Note: Converter functions always get called with a string, no matter under which data type you sent the value to SQLite.

def convert_point(s):
    x, y = map(float, s.split(';'))
    return Point(x, y)

Now you need to make the sqlite3 module know that what you select from the database is actually a point. There are two ways of
doing this:

- Implicitly via the declared type
- Explicitly via the column name

Both ways are described in section Module functions and constants, in the entries for the constants `PARSE_DECLTYPES` and `PARSE_COLNAMES`.

The following example illustrates both approaches.

```python
import sqlite3

class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "(%f;%f)" % (self.x, self.y)

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

def convert_point(s):
    x, y = map(float, s.split(";"))
    return Point(x, y)

# Register the adapter
sqlite3.register_adapter(Point, adapt_point)

# Register the converter
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

# 1) Using declared types
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p point)"

cur.execute("insert into test(p) values (?)", (p,))
cur.execute("select p from test")
print "with declared types:", cur.fetchone()[0]
cur.close()
```
11.13.5.4. Default adapters and converters

There are default adapters for the date and datetime types in the datetime module. They will be sent as ISO dates/ISO timestamps to SQLite.

The default converters are registered under the name “date” for \texttt{datetime.date} and under the name “timestamp” for \texttt{datetime.datetime}.

This way, you can use date/timestamps from Python without any additional fiddling in most cases. The format of the adapters is also compatible with the experimental SQLite date/time functions.

The following example demonstrates this.

```python
import sqlite3
import datetime

con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)")
today = datetime.date.today()
now = datetime.datetime.now()
```
cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
row = cur.fetchone()
print today, "=>", row[0], type(row[0])
print now, "=>", row[1], type(row[1])

cur.execute('select current_date as "d [date]", current_timestamp as "ts [timestamp]"
row = cur.fetchone()
print "current_date", row[0], type(row[0])
print "current_timestamp", row[1], type(row[1])

If a timestamp stored in SQLite has a fractional part longer than 6
numbers, its value will be truncated to microsecond precision by the
timestamp converter.
11.13.6. Controlling Transactions

By default, the `sqlite3` module opens transactions implicitly before a Data Modification Language (DML) statement (i.e. `INSERT/UPDATE/DELETE/REPLACE`), and commits transactions implicitly before a non-DML, non-query statement (i.e. anything other than `SELECT` or the aforementioned).

So if you are within a transaction and issue a command like `CREATE TABLE ...`, `VACUUM`, `PRAGMA`, the `sqlite3` module will commit implicitly before executing that command. There are two reasons for doing that. The first is that some of these commands don't work within transactions. The other reason is that sqlite3 needs to keep track of the transaction state (if a transaction is active or not).

You can control which kind of `BEGIN` statements sqlite3 implicitly executes (or none at all) via the `isolation_level` parameter to the `connect()` call, or via the `isolation_level` property of connections.

If you want **autocommit mode**, then set `isolation_level` to `None`.

Otherwise leave it at its default, which will result in a plain “BEGIN” statement, or set it to one of SQLite’s supported isolation levels: “DEFERRED”, “IMMEDIATE” or “EXCLUSIVE”.
11.13.7. Using sqlite3 efficiently

11.13.7.1. Using shortcut methods

Using the nonstandard `execute()`, `executemany()` and `executescript()` methods of the `Connection` object, your code can be written more concisely because you don't have to create the (often superfluous) `Cursor` objects explicitly. Instead, the `Cursor` objects are created implicitly and these shortcut methods return the cursor objects. This way, you can execute a `SELECT` statement and iterate over it directly using only a single call on the `Connection` object.

```python
import sqlite3

persons = [
    ("Hugo", "Boss"),
    ("Calvin", "Klein")
]

con = sqlite3.connect(":memory:")

# Create the table
con.execute("create table person(firstname, lastname)")

# Fill the table
con.executemany("insert into person(firstname, lastname) values (?, ?)"

# Print the table contents
for row in con.execute("select firstname, lastname from person"
    print row

print "I just deleted", con.execute("delete from person").rowcount
```

11.13.7.2. Accessing columns by name instead of by index
One useful feature of the sqlite3 module is the built-in sqlite3.Row class designed to be used as a row factory.

Rows wrapped with this class can be accessed both by index (like tuples) and case-insensitively by name:

```python
import sqlite3

con = sqlite3.connect(":memory:"
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("select 'John' as name, 42 as age")
for row in cur:
    assert row[0] == row["name"]
    assert row["name"] == row["nAmE"]
    assert row[1] == row["age"]
    assert row[1] == row["AgE"]
```

11.13.7.3. Using the connection as a context manager

New in version 2.6.

Connection objects can be used as context managers that automatically commit or rollback transactions. In the event of an exception, the transaction is rolled back; otherwise, the transaction is committed:

```python
import sqlite3

con = sqlite3.connect(":memory:"
con.execute("create table person (id integer primary key, firstname varchar unique)
#	Successful, con.commit() is called automatically afterwards
with con:
    con.execute("insert into person(firstname) values (?)", ("J")
#	con.rollback() is called after the with block finishes with a
# exception is still raised and must be caught
```
try:
    with con:
        con.execute("insert into person(firstname) values (?)",
    except sqlite3.IntegrityError:
        print "couldn't add Joe twice"
11.13.8. Common issues

11.13.8.1. Multithreading

Older SQLite versions had issues with sharing connections between threads. That’s why the Python module disallows sharing connections and cursors between threads. If you still try to do so, you will get an exception at runtime.

The only exception is calling the `interrupt()` method, which only makes sense to call from a different thread.

Footnotes

(1, 2) The `sqlite3` module is not built with loadable extension support by default, because some platforms (notably Mac OS X) have SQLite libraries which are compiled without this feature. To get loadable extension support, you must modify `setup.py` and remove the line that sets `SQLITE_OMIT_LOAD_EXTENSION`. 

[1]
12. Data Compression and Archiving

The modules described in this chapter support data compression with the zlib, gzip, and bzip2 algorithms, and the creation of ZIP- and tar-format archives. See also Archiving operations provided by the `shutil` module.

- **12.1. zlib** — Compression compatible with gzip
- **12.2. gzip** — Support for gzip files
  - 12.2.1. Examples of usage
- **12.3. bzip2** — Compression compatible with bzip2
  - 12.3.1. (De)compression of files
  - 12.3.2. Sequential (de)compression
  - 12.3.3. One-shot (de)compression
- **12.4. zipfile** — Work with ZIP archives
  - 12.4.1. ZipFile Objects
  - 12.4.2. PyZipFile Objects
  - 12.4.3. ZipInfo Objects
  - 12.4.4. Command-Line Interface
    - 12.4.4.1. Command-line options
- **12.5. tarfile** — Read and write tar archive files
  - 12.5.1. TarFile Objects
  - 12.5.2. TarInfo Objects
  - 12.5.3. Examples
  - 12.5.4. Supported tar formats
  - 12.5.5. Unicode issues
12.1. **zlib** — Compression compatible with **gzip**

For applications that require data compression, the functions in this module allow compression and decompression, using the zlib library. The zlib library has its own home page at [http://www.zlib.net](http://www.zlib.net). There are known incompatibilities between the Python module and versions of the zlib library earlier than 1.1.3; 1.1.3 has a security vulnerability, so we recommend using 1.1.4 or later.

zlib’s functions have many options and often need to be used in a particular order. This documentation doesn’t attempt to cover all of the permutations; consult the zlib manual at [http://www.zlib.net/manual.html](http://www.zlib.net/manual.html) for authoritative information.

For reading and writing `.gz` files see the **gzip** module.

The available exception and functions in this module are:

- **exception** `zlib.error`
  
  Exception raised on compression and decompression errors.

- `zlib.adler32(data[, value])`
  
  Computes an Adler-32 checksum of `data`. (An Adler-32 checksum is almost as reliable as a CRC32 but can be computed much more quickly.) If `value` is present, it is used as the starting value of the checksum; otherwise, a fixed default value is used. This allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.
This function always returns an integer object.

**Note:** To generate the same numeric value across all Python versions and platforms use adler32(data) & 0xffffffff. If you are only using the checksum in packed binary format this is not necessary as the return value is the correct 32bit binary representation regardless of sign.

*Changed in version 2.6:* The return value is in the range [-2**31, 2**31-1] regardless of platform. In older versions the value is signed on some platforms and unsigned on others.

*Changed in version 3.0:* The return value is unsigned and in the range [0, 2**32-1] regardless of platform.

zlib.```compress```(string[, level])

Compresses the data in `string`, returning a string contained compressed data. `level` is an integer from 0 to 9 controlling the level of compression; 1 is fastest and produces the least compression, 9 is slowest and produces the most. 0 is no compression. The default value is 6. Raises the error exception if any error occurs.

zlib.```compressobj```([level[, method[, wbits[, memlevel[, strategy]]]]]])

Returns a compression object, to be used for compressing data streams that won't fit into memory at once. `level` is an integer from 0 to 9 or -1, controlling the level of compression; 1 is fastest and produces the least compression, 9 is slowest and produces the most. 0 is no compression. The default value is -1 (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION represents a default compromise between speed and compression (currently equivalent to level 6).
**method** is the compression algorithm. Currently, the only supported value is **DEFLATED**.

The **wbits** argument controls the size of the history buffer (or the “window size”) used when compressing data, and whether a header and trailer is included in the output. It can take several ranges of values. The default is 15.

- +9 to +15: The base-two logarithm of the window size, which therefore ranges between 512 and 32768. Larger values produce better compression at the expense of greater memory usage. The resulting output will include a zlib-specific header and trailer.
- −9 to −15: Uses the absolute value of **wbits** as the window size logarithm, while producing a raw output stream with no header or trailing checksum.
- +25 to +31 = 16 + (9 to 15): Uses the low 4 bits of the value as the window size logarithm, while including a basic **gzip** header and trailing checksum in the output.

**memlevel** controls the amount of memory used for internal compression state. Valid values range from 1 to 9. Higher values using more memory, but are faster and produce smaller output. The default is 8.

**strategy** is used to tune the compression algorithm. Possible values are **Z_DEFAULT_STRATEGY**, **Z_FILTERED**, and **Z_HUFFMAN_ONLY**. The default is **Z_DEFAULT_STRATEGY**.

**zlib.crc32(data[, value])**

Computes a CRC (Cyclic Redundancy Check) checksum of *data*. If *value* is present, it is used as the starting value of the checksum; otherwise, a fixed default value is used. This allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should
not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

This function always returns an integer object.

**Note:** To generate the same numeric value across all Python versions and platforms use crc32(data) & 0xffffffff. If you are only using the checksum in packed binary format this is not necessary as the return value is the correct 32bit binary representation regardless of sign.

*Changed in version 2.6:* The return value is in the range \([-2^{31}, 2^{31}-1]\) regardless of platform. In older versions the value would be signed on some platforms and unsigned on others.

*Changed in version 3.0:* The return value is unsigned and in the range \([0, 2^{32}-1]\) regardless of platform.

`zlib.decompress(string[, wbits[, bufsize]])`

Decompresses the data in `string`, returning a string containing the uncompressed data. The `wbits` parameter depends on the format of `string`, and is discussed further below. If `bufsize` is given, it is used as the initial size of the output buffer. Raises the `error` exception if any error occurs.

The `wbits` parameter controls the size of the history buffer (or “window size”), and what header and trailer format is expected. It is similar to the parameter for `compressobj()`, but accepts more ranges of values:

- +8 to +15: The base-two logarithm of the window size. The input must include a zlib header and trailer.
- 0: Automatically determine the window size from the zlib header. Only supported since zlib 1.2.3.5.
-8 to -15: Uses the absolute value of \textit{wbits} as the window size logarithm. The input must be a raw stream with no header or trailer.
+24 to +31 = 16 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm. The input must include a gzip header and trailer.
+40 to +47 = 32 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm, and automatically accepts either the zlib or gzip format.

When decompressing a stream, the window size must not be smaller than the size originally used to compress the stream; using a too-small value may result in an \texttt{error} exception. The default \textit{wbits} value is 15, which corresponds to the largest window size and requires a zlib header and trailer to be included.

\textit{bufsize} is the initial size of the buffer used to hold decompressed data. If more space is required, the buffer size will be increased as needed, so you don’t have to get this value exactly right; tuning it will only save a few calls to \texttt{malloc()}. The default size is 16384.

\texttt{zlib.decompressobj([\textit{wbits}])}

Returns a decompression object, to be used for decompressing data streams that won’t fit into memory at once.

The \textit{wbits} parameter controls the size of the history buffer (or the “window size”), and what header and trailer format is expected. It has the same meaning as described for \texttt{decompress()}. Compression objects support the following methods:

\texttt{Compress.compress(string)}

Compress \textit{string}, returning a string containing compressed data for at least part of the data in \textit{string}. This data should be
concatenated to the output produced by any preceding calls to the `compress()` method. Some input may be kept in internal buffers for later processing.

**Compress. `flush([mode])`**

All pending input is processed, and a string containing the remaining compressed output is returned. `mode` can be selected from the constants `Z_SYNC_FLUSH`, `Z_FULL_FLUSH`, or `Z_FINISH`, defaulting to `Z_FINISH`. `Z_SYNC_FLUSH` and `Z_FULL_FLUSH` allow compressing further strings of data, while `Z_FINISH` finishes the compressed stream and prevents compressing any more data. After calling `flush()` with `mode` set to `Z_FINISH`, the `compress()` method cannot be called again; the only realistic action is to delete the object.

**Compress. `copy()`**

Returns a copy of the compression object. This can be used to efficiently compress a set of data that share a common initial prefix.

*New in version 2.5.*

Decompression objects support the following methods, and two attributes:

**Decompress. `unused_data`**

A string which contains any bytes past the end of the compressed data. That is, this remains "" until the last byte that contains compression data is available. If the whole string turned out to contain compressed data, this is "", the empty string.

The only way to determine where a string of compressed data ends is by actually decompressing it. This means that when compressed data is contained part of a larger file, you can only
find the end of it by reading data and feeding it followed by some non-empty string into a decompression object's `decompress()` method until the `unused_data` attribute is no longer the empty string.

**Decompress.** `unconsumed_tail`

A string that contains any data that was not consumed by the last `decompress()` call because it exceeded the limit for the uncompressed data buffer. This data has not yet been seen by the zlib machinery, so you must feed it (possibly with further data concatenated to it) back to a subsequent `decompress()` method call in order to get correct output.

**Decompress.** `decompress(string[, max_length])`

Decompress string, returning a string containing the uncompressed data corresponding to at least part of the data in string. This data should be concatenated to the output produced by any preceding calls to the `decompress()` method. Some of the input data may be preserved in internal buffers for later processing.

If the optional parameter `max_length` is non-zero then the return value will be no longer than `max_length`. This may mean that not all of the compressed input can be processed; and unconsumed data will be stored in the attribute `unconsumed_tail`. This string must be passed to a subsequent call to `decompress()` if decompression is to continue. If `max_length` is not supplied then the whole input is decompressed, and `unconsumed_tail` is an empty string.

**Decompress.** `flush([length])`

All pending input is processed, and a string containing the remaining uncompressed output is returned. After calling
flush(), the decompress() method cannot be called again; the only realistic action is to delete the object.

The optional parameter length sets the initial size of the output buffer.

Decompress.copy()

Returns a copy of the decompression object. This can be used to save the state of the decompressor midway through the data stream in order to speed up random seeks into the stream at a future point.

New in version 2.5.

See also:

Module gzip
  Reading and writing gzip-format files.

http://www.zlib.net
  The zlib library home page.

http://www.zlib.net/manual.html
  The zlib manual explains the semantics and usage of the library's many functions.
12.2. gzip — Support for gzip files

Source code: Lib/gzip.py

This module provides a simple interface to compress and decompress files just like the GNU programs gzip and gunzip would.

The data compression is provided by the zlib module.

The gzip module provides the GzipFile class which is modeled after Python's File Object. The GzipFile class reads and writes gzip-format files, automatically compressing or decompressing the data so that it looks like an ordinary file object.

Note that additional file formats which can be decompressed by the gzip and gunzip programs, such as those produced by compress and pack, are not supported by this module.

The module defines the following items:

```
class GzipFile([filename[, mode[, compresslevel[, fileobj[, mtime]]]]])
```

Constructor for the GzipFile class, which simulates most of the methods of a file object, with the exception of the readinto() and truncate() methods. At least one of fileobj and filename must be given a non-trivial value.

The new class instance is based on fileobj, which can be a regular file, a StringIO object, or any other object which simulates a file. It defaults to None, in which case filename is opened to provide a file object.
When `fileobj` is not `None`, the `filename` argument is only used to be included in the `gzip` file header, which may include the original filename of the uncompressed file. It defaults to the filename of `fileobj`, if discernible; otherwise, it defaults to the empty string, and in this case the original filename is not included in the header.

The `mode` argument can be any of `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, or `'wb'`, depending on whether the file will be read or written. The default is the mode of `fileobj` if discernible; otherwise, the default is `'rb'`. If not given, the ‘b’ flag will be added to the mode to ensure the file is opened in binary mode for cross-platform portability.

The `compresslevel` argument is an integer from 0 to 9 controlling the level of compression; 1 is fastest and produces the least compression, and 9 is slowest and produces the most compression. 0 is no compression. The default is 9.

The `mtime` argument is an optional numeric timestamp to be written to the stream when compressing. All `gzip` compressed streams are required to contain a timestamp. If omitted or `None`, the current time is used. This module ignores the timestamp when decompressing; however, some programs, such as `gunzip`, make use of it. The format of the timestamp is the same as that of the return value of `time.time()` and of the `st_mtime` attribute of the object returned by `os.stat()`.

Calling a `GzipFile` object’s `close()` method does not close `fileobj`, since you might wish to append more material after the compressed data. This also allows you to pass a `StringIO` object opened for writing as `fileobj`, and retrieve the resulting memory buffer using the `StringIO` object’s `getvalue()` method.
GzipFile supports iteration and the with statement.

*Changed in version 2.7:* Support for the with statement was added.

*Changed in version 2.7:* Support for zero-padded files was added.

*New in version 2.7:* The mtime argument.

```python
gzip.open(filename[, mode[, compresslevel]])
```
This is a shorthand for `GzipFile(filename, mode, compresslevel)`. The filename argument is required; mode defaults to 'rb' and compresslevel defaults to 9.
12.2.1. Examples of usage

Example of how to read a compressed file:

```python
import gzip
with gzip.open('file.txt.gz', 'rb') as f:
    file_content = f.read()
```

Example of how to create a compressed GZIP file:

```python
import gzip
content = "Lots of content here"
with gzip.open('file.txt.gz', 'wb') as f:
    f.write(content)
```

Example of how to GZIP compress an existing file:

```python
import gzip
import shutil
with open('file.txt', 'rb') as f_in, gzip.open('file.txt.gz', 'wb') as f:
    shutil.copyfileobj(f_in, f)
```

See also:

Module `zlib`

The basic data compression module needed to support the `gzip` file format.
12.3. **bz2 — Compression compatible with bzip2**

*New in version 2.3.*

This module provides a comprehensive interface for the bz2 compression library. It implements a complete file interface, one-shot (de)compression functions, and types for sequential (de)compression.

Here is a summary of the features offered by the bz2 module:

- **BZ2File** class implements a complete file interface, including `readline()`, `readlines()`, `writelines()`, `seek()`, etc;
- **BZ2File** class implements emulated `seek()` support;
- **BZ2File** class implements universal newline support;
- **BZ2File** class offers an optimized line iteration using the readahead algorithm borrowed from file objects;
- Sequential (de)compression supported by **BZ2Compressor** and **BZ2Decompressor** classes;
- One-shot (de)compression supported by `compress()` and `decompress()` functions;
- Thread safety uses individual locking mechanism.
12.3.1. (De)compression of files

Handling of compressed files is offered by the `BZ2File` class.

```python
class bz2.BZ2File(filename[, mode[, buffering[, compresslevel]]])
```

Open a bz2 file. Mode can be either 'r' or 'w', for reading (default) or writing. When opened for writing, the file will be created if it doesn’t exist, and truncated otherwise. If `buffering` is given, 0 means unbuffered, and larger numbers specify the buffer size; the default is 0. If `compresslevel` is given, it must be a number between 1 and 9; the default is 9. Add a 'u' to mode to open the file for input in universal newlines mode. Any line ending in the input file will be seen as a '\n' in Python. Also, a file so opened gains the attribute `newlines`; the value for this attribute is one of `None` (no newline read yet), '\r', '\n', '\r\n' or a tuple containing all the newline types seen. Universal newlines are available only when reading. Instances support iteration in the same way as normal `file` instances.

`BZ2File` supports the `with` statement.

*Changed in version 2.7:* Support for the `with` statement was added.

**Note:** This class does not support input files containing multiple streams (such as those produced by the `pbzip2` tool). When reading such an input file, only the first stream will be accessible. If you require support for multi-stream files, consider using the third-party `bz2file` module (available from PyPI). This module provides a backport of Python 3.3’s `BZ2File` class, which does support multi-stream files.
**close()**
Close the file. Sets data attribute `closed` to true. A closed file cannot be used for further I/O operations. `close()` may be called more than once without error.

**read([size])**
Read at most size uncompressed bytes, returned as a string. If the size argument is negative or omitted, read until EOF is reached.

**readline([size])**
Return the next line from the file, as a string, retaining newline. A non-negative size argument limits the maximum number of bytes to return (an incomplete line may be returned then). Return an empty string at EOF.

**readlines([size])**
Return a list of lines read. The optional size argument, if given, is an approximate bound on the total number of bytes in the lines returned.

**xreadlines()**
For backward compatibility. `BZ2File` objects now include the performance optimizations previously implemented in the `xreadlines` module.

*Deprecated since version 2.3:* This exists only for compatibility with the method by this name on `file` objects, which is deprecated. Use `for line in file` instead.

**seek(offset[, whence])**
Move to new file position. Argument `offset` is a byte count. Optional argument `whence` defaults to `os.SEEK_SET` or 0.
(offset from start of file; offset should be $\geq 0$); other values are \texttt{os.SEEK_CUR} or 1 (move relative to current position; offset can be positive or negative), and \texttt{os.SEEK_END} or 2 (move relative to end of file; offset is usually negative, although many platforms allow seeking beyond the end of a file).

Note that seeking of bz2 files is emulated, and depending on the parameters the operation may be extremely slow.

\begin{itemize}
\item \texttt{tell()}
\end{itemize}

Return the current file position, an integer (may be a long integer).

\begin{itemize}
\item \texttt{write(data)}
\end{itemize}

Write string \texttt{data} to file. Note that due to buffering, \texttt{close()} may be needed before the file on disk reflects the data written.

\begin{itemize}
\item \texttt{writelines(sequence_of_strings)}
\end{itemize}

Write the sequence of strings to the file. Note that newlines are not added. The sequence can be any iterable object producing strings. This is equivalent to calling \texttt{write()} for each string.
12.3.2. Sequential (de)compression

Sequential compression and decompression is done using the classes `BZ2Compressor` and `BZ2Decompressor`.

```python
class bz2.BZ2Compressor([compresslevel])
    Create a new compressor object. This object may be used to compress data sequentially. If you want to compress data in one shot, use the `compress()` function instead. The `compresslevel` parameter, if given, must be a number between 1 and 9; the default is 9.

    `compress(data)`
    Provide more data to the compressor object. It will return chunks of compressed data whenever possible. When you've finished providing data to compress, call the `flush()` method to finish the compression process, and return what is left in internal buffers.

    `flush()`
    Finish the compression process and return what is left in internal buffers. You must not use the compressor object after calling this method.

class bz2.BZ2Decompressor
    Create a new decompressor object. This object may be used to decompress data sequentially. If you want to decompress data in one shot, use the `decompress()` function instead.

    `decompress(data)`
    Provide more data to the decompressor object. It will return chunks of decompressed data whenever possible. If you try to
decompress data after the end of stream is found, `EOFError` will be raised. If any data was found after the end of stream, it'll be ignored and saved in `unused_data` attribute.
12.3.3. One-shot (de)compression

One-shot compression and decompression is provided through the `compress()` and `decompress()` functions.

```
bz2.compress(data[, compresslevel])
```
Compress `data` in one shot. If you want to compress data sequentially, use an instance of `BZ2Compressor` instead. The `compresslevel` parameter, if given, must be a number between 1 and 9; the default is 9.

```
bz2.decompress(data)
```
Decompress `data` in one shot. If you want to decompress data sequentially, use an instance of `BZ2Decompressor` instead.
The ZIP file format is a common archive and compression standard. This module provides tools to create, read, write, append, and list a ZIP file. Any advanced use of this module will require an understanding of the format, as defined in PKZIP Application Note.

This module does not currently handle multi-disk ZIP files. It can handle ZIP files that use the ZIP64 extensions (that is ZIP files that are more than 4 GByte in size). It supports decryption of encrypted files in ZIP archives, but it currently cannot create an encrypted file. Decryption is extremely slow as it is implemented in native Python rather than C.

The module defines the following items:

**exception** zipfile.**BadZipfile**

The error raised for bad ZIP files (old name: zipfile.error).

**exception** zipfile.**LargeZipFile**

The error raised when a ZIP file would require ZIP64 functionality but that has not been enabled.

**class** zipfile.**ZipFile**

The class for reading and writing ZIP files. See section ZipFile Objects for constructor details.

**class** zipfile.**PyZipFile**
Class for creating ZIP archives containing Python libraries.

```python
class zipfile.ZipInfo([filename[, date_time]])
```
Class used to represent information about a member of an archive. Instances of this class are returned by the `getinfo()` and `infolist()` methods of `ZipFile` objects. Most users of the `zipfile` module will not need to create these, but only use those created by this module. `filename` should be the full name of the archive member, and `date_time` should be a tuple containing six fields which describe the time of the last modification to the file; the fields are described in section `ZipInfo Objects`.

```python
zipfile.is_zipfile(filename)
```
Returns `True` if `filename` is a valid ZIP file based on its magic number, otherwise returns `False`. `filename` may be a file or file-like object too.

*Changed in version 2.7:* Support for file and file-like objects.

```python
zipfile.ZIP_STORED
```
The numeric constant for an uncompressed archive member.

```python
zipfile.ZIP_DEFLATED
```
The numeric constant for the usual ZIP compression method. This requires the `zlib` module. No other compression methods are currently supported.

**See also:**

**PKZIP Application Note**
Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

**Info-ZIP Home Page**
Information about the Info-ZIP project's ZIP archive programs
and development libraries.
class zipfile.ZipFile(file[, mode[, compression[, allowZip64]]]])

Open a ZIP file, where file can be either a path to a file (a string) or a file-like object. The mode parameter should be 'r' to read an existing file, 'w' to truncate and write a new file, or 'a' to append to an existing file. If mode is 'a' and file refers to an existing ZIP file, then additional files are added to it. If file does not refer to a ZIP file, then a new ZIP archive is appended to the file. This is meant for adding a ZIP archive to another file (such as python.exe).

*Changed in version 2.6:* If mode is 'a' and the file does not exist at all, it is created.

compression is the ZIP compression method to use when writing the archive, and should be ZIP_STORED or ZIP_DEFLATED; unrecognized values will cause RuntimeError to be raised. If ZIP_DEFLATED is specified but the zlib module is not available, RuntimeError is also raised. The default is ZIP_STORED. If allowZip64 is True zipfile will create ZIP files that use the ZIP64 extensions when the zipfile is larger than 2 GB. If it is false (the default) zipfile will raise an exception when the ZIP file would require ZIP64 extensions. ZIP64 extensions are disabled by default because the default zip and unzip commands on Unix (the InfoZIP utilities) don’t support these extensions.

*Changed in version 2.7.1:* If the file is created with mode 'a' or 'w' and then closed without adding any files to the archive, the appropriate ZIP structures for an empty archive will be written to the file.
ZipFile is also a context manager and therefore supports the `with` statement. In the example, `myzip` is closed after the `with` statement’s suite is finished—even if an exception occurs:

```python
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

*New in version 2.7:* Added the ability to use `ZipFile` as a context manager.

`ZipFile.close()`  
Close the archive file. You must call `close()` before exiting your program or essential records will not be written.

`ZipFile.getinfo(name)`  
Return a `ZipInfo` object with information about the archive member `name`. Calling `getinfo()` for a name not currently contained in the archive will raise a `KeyError`.

`ZipFile.infolist()`  
Return a list containing a `ZipInfo` object for each member of the archive. The objects are in the same order as their entries in the actual ZIP file on disk if an existing archive was opened.

`ZipFile.namelist()`  
Return a list of archive members by name.

`ZipFile.open(name[, mode[, pwd]])`  
Extract a member from the archive as a file-like object (ZipExtFile). `name` is the name of the file in the archive, or a `ZipInfo` object. The `mode` parameter, if included, must be one of the following: `'r'` (the default), `'U'`, or `'rU'`. Choosing `'U'` or `'rU'` will enable universal newline support in the read-only object. `pwd` is the password used for encrypted files. Calling `open()` on a
closed ZipFile will raise a `RuntimeError`.

**Note:** The file-like object is read-only and provides the following methods: `read()`, `readline()`, `readlines()`, `__iter__()`, `next()`.

**Note:** If the ZipFile was created by passing in a file-like object as the first argument to the constructor, then the object returned by `open()` shares the ZipFile’s file pointer. Under these circumstances, the object returned by `open()` should not be used after any additional operations are performed on the ZipFile object. If the ZipFile was created by passing in a string (the filename) as the first argument to the constructor, then `open()` will create a new file object that will be held by the ZipExtFile, allowing it to operate independently of the ZipFile.

**Note:** The `open()`, `read()` and `extract()` methods can take a filename or a `ZipInfo` object. You will appreciate this when trying to read a ZIP file that contains members with duplicate names.

*New in version 2.6.*

```python
ZipFile.extract(member[, path[, pwd]])
```

Extract a member from the archive to the current working directory; `member` must be its full name or a `ZipInfo` object. Its file information is extracted as accurately as possible. `path` specifies a different directory to extract to. `member` can be a filename or a `ZipInfo` object. `pwd` is the password used for encrypted files.

Returns the normalized path created (a directory or new file).

*New in version 2.6.*
Note: If a member filename is an absolute path, a drive/UNC sharepoint and leading (back)slashes will be stripped, e.g.: ///foo/bar becomes foo/bar on Unix, and C:\foo\bar becomes foo\bar on Windows. And all ".." components in a member filename will be removed, e.g.: ../../foo/../../bar becomes foo../ba..r. On Windows illegal characters (:, <, >, |, ", ?, and *) replaced by underscore (_).

```
ZipFile.extractall([path[, members[, pwd]]])
```

Extract all members from the archive to the current working directory. `path` specifies a different directory to extract to. `members` is optional and must be a subset of the list returned by `namelist()`. `pwd` is the password used for encrypted files.

**Warning:** Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of `path`, e.g. members that have absolute filenames starting with "/" or filenames with two dots "..".

*Changed in version 2.7.4:* The zipfile module attempts to prevent that. See `extract()` note.

*New in version 2.6.*

```
ZipFile.printdir()
```

Print a table of contents for the archive to `sys.stdout`.

```
ZipFile.setpassword(pwd)
```

Set `pwd` as default password to extract encrypted files.

*New in version 2.6.*

```
ZipFile.read(name[, pwd])
```
Return the bytes of the file *name* in the archive. *name* is the name of the file in the archive, or a `ZipInfo` object. The archive must be open for read or append. *pwd* is the password used for encrypted files and, if specified, it will override the default password set with `setpassword()`. Calling `read()` on a closed ZipFile will raise a `RuntimeError`.

*Changed in version 2.6:* *pwd* was added, and *name* can now be a `ZipInfo` object.

**ZipFile.**`testzip()`
Read all the files in the archive and check their CRC's and file headers. Return the name of the first bad file, or else return `None`. Calling `testzip()` on a closed ZipFile will raise a `RuntimeError`.

**ZipFile.**`write(filename[, arcname[, compress_type]])`
Write the file named `filename` to the archive, giving it the archive name `arcname` (by default, this will be the same as `filename`, but without a drive letter and with leading path separators removed). If given, `compress_type` overrides the value given for the `compression` parameter to the constructor for the new entry. The archive must be open with mode `'w'` or `'a'` – calling `write()` on a ZipFile created with mode `'r'` will raise a `RuntimeError`. Calling `write()` on a closed ZipFile will raise a `RuntimeError`.

**Note:** There is no official file name encoding for ZIP files. If you have unicode file names, you must convert them to byte strings in your desired encoding before passing them to `write()`. WinZip interprets all file names as encoded in CP437, also known as DOS Latin.

**Note:** Archive names should be relative to the archive root, that is, they should not start with a path separator.
Note: If arcname (or filename, if arcname is not given) contains a null byte, the name of the file in the archive will be truncated at the null byte.

ZipFile.writestr(zinfo_or_arcname, bytes[, compress_type])
Write the string bytes to the archive; zinfo_or_arcname is either the file name it will be given in the archive, or a ZipInfo instance. If it’s an instance, at least the filename, date, and time must be given. If it’s a name, the date and time is set to the current date and time. The archive must be opened with mode 'w' or 'a' – calling writestr() on a ZipFile created with mode 'r' will raise a RuntimeError. Calling writestr() on a closed ZipFile will raise a RuntimeError.

If given, compress_type overrides the value given for the compression parameter to the constructor for the new entry, or in the zinfo_or_arcname (if that is a ZipInfo instance).

Note: When passing a ZipInfo instance as the zinfo_or_arcname parameter, the compression method used will be that specified in the compress_type member of the given ZipInfo instance. By default, the ZipInfo constructor sets this member to ZIP_STORED.

Changed in version 2.7: The compress_type argument.

The following data attributes are also available:

ZipFile.debug
The level of debug output to use. This may be set from 0 (the default, no output) to 3 (the most output). Debugging information is written to sys.stdout.
ZipFile.comment

The comment text associated with the ZIP file. If assigning a comment to a ZipFile instance created with mode ‘a’ or ‘w’, this should be a string no longer than 65535 bytes. Comments longer than this will be truncated in the written archive when close() is called.
12.4.2. PyZipFile Objects

The PyZipFile constructor takes the same parameters as the ZipFile constructor. Instances have one method in addition to those of ZipFile objects.

PyZipFile.writepy(pathname[, basename])

Search for files *.py and add the corresponding file to the archive. The corresponding file is a *.pyo file if available, else a *.pyc file, compiling if necessary. If the pathname is a file, the filename must end with .py, and just the (corresponding *.py[co]) file is added at the top level (no path information). If the path is a file that does not end with .py, a RuntimeError will be raised. If it is a directory, and the directory is not a package directory, then all the files *.py[co] are added at the top level. If the directory is a package directory, then all *.py[co] are added under the package name as a file path, and if any subdirectories are package directories, all of these are added recursively. basename is intended for internal use only. The writepy() method makes archives with file names like this:

```
string.pyc  # Top level name
test/__init__.pyc  # Package directory
    # Module test
test/test_support.pyc  # Subpackage direc
test/bogus/__init__.pyc  # Submodule test.b
test/bogus/myfile.pyc
```
12.4.3. ZipInfo Objects

Instances of the `ZipInfo` class are returned by the `getinfo()` and `infolist()` methods of `ZipFile` objects. Each object stores information about a single member of the ZIP archive.

Instances have the following attributes:

`ZipInfo.filename`
   Name of the file in the archive.

`ZipInfo.date_time`
   The time and date of the last modification to the archive member. This is a tuple of six values:

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Year (&gt;= 1980)</td>
</tr>
<tr>
<td>1</td>
<td>Month (one-based)</td>
</tr>
<tr>
<td>2</td>
<td>Day of month (one-based)</td>
</tr>
<tr>
<td>3</td>
<td>Hours (zero-based)</td>
</tr>
<tr>
<td>4</td>
<td>Minutes (zero-based)</td>
</tr>
<tr>
<td>5</td>
<td>Seconds (zero-based)</td>
</tr>
</tbody>
</table>

**Note:** The ZIP file format does not support timestamps before 1980.

`ZipInfo.compress_type`
   Type of compression for the archive member.

`ZipInfo.comment`
   Comment for the individual archive member.

`ZipInfo.extra`
Expansion field data. The PKZIP Application Note contains some comments on the internal structure of the data contained in this string.

**ZipInfo**.\texttt{create_system}  
System which created ZIP archive.

**ZipInfo**.\texttt{create_version}  
PKZIP version which created ZIP archive.

**ZipInfo**.\texttt{extract_version}  
PKZIP version needed to extract archive.

**ZipInfo**.\texttt{reserved}  
Must be zero.

**ZipInfo**.\texttt{flag_bits}  
ZIP flag bits.

**ZipInfo**.\texttt{volume}  
Volume number of file header.

**ZipInfo**.\texttt{internal_attr}  
Internal attributes.

**ZipInfo**.\texttt{external_attr}  
External file attributes.

**ZipInfo**.\texttt{header_offset}  
Byte offset to the file header.

**ZipInfo**.\texttt{CRC}  
CRC-32 of the uncompressed file.

**ZipInfo**.\texttt{compress_size}  
Size of the compressed data.

**ZipInfo**.\texttt{file_size}
Size of the uncompressed file.
12.4.4. Command-Line Interface

The `zipfile` module provides a simple command-line interface to interact with ZIP archives.

If you want to create a new ZIP archive, specify its name after the `-c` option and then list the filename(s) that should be included:

```bash
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

Passing a directory is also acceptable:

```bash
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

If you want to extract a ZIP archive into the specified directory, use the `-e` option:

```bash
$ python -m zipfile -e monty.zip target-dir/
```

For a list of the files in a ZIP archive, use the `-l` option:

```bash
$ python -m zipfile -l monty.zip
```

12.4.4.1. Command-line options

- `-l <zipfile>`
  List files in a zipfile.

- `-c <zipfile> <source1> ... <sourceN>`
  Create zipfile from source files.

- `-e <zipfile> <output_dir>`
  Extract zipfile into target directory.
-t < zipfile>

Test whether the zipfile is valid or not.
12.5. tarfile — Read and write tar archive files

New in version 2.3.

Source code: Lib/tarfile.py

The tarfile module makes it possible to read and write tar archives, including those using gzip or bz2 compression. Use the zipfile module to read or write .zip files, or the higher-level functions in shutil.

Some facts and figures:

- reads and writes gzip and bz2 compressed archives if the respective modules are available.

- read/write support for the POSIX.1-1988 (ustar) format.

- read/write support for the GNU tar format including longname and longlink extensions, read-only support for the sparse extension.

- read/write support for the POSIX.1-2001 (pax) format.

  New in version 2.6.

- handles directories, regular files, hardlinks, symbolic links, fifos, character devices and block devices and is able to acquire and restore file information like timestamp, access permissions and owner.

tarfile.open(name=None, mode='r', fileobj=None, bufsize=10240, **kwargs)
Return a TarFile object for the pathname name. For detailed information on TarFile objects and the keyword arguments that are allowed, see TarFile Objects.

mode has to be a string of the form 'filemode[:compression]', it defaults to 'r'. Here is a full list of mode combinations:

<table>
<thead>
<tr>
<th>mode</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r' or 'r:*'</td>
<td>Open for reading with transparent compression (recommended).</td>
</tr>
<tr>
<td>'r:'</td>
<td>Open for reading exclusively without compression.</td>
</tr>
<tr>
<td>'r:gz'</td>
<td>Open for reading with gzip compression.</td>
</tr>
<tr>
<td>'r:bz2'</td>
<td>Open for reading with bzip2 compression.</td>
</tr>
<tr>
<td>'a' or 'a:'</td>
<td>Open for appending with no compression. The file is created if it does not exist.</td>
</tr>
<tr>
<td>'w' or 'w:'</td>
<td>Open for uncompressed writing.</td>
</tr>
<tr>
<td>'w:gz'</td>
<td>Open for gzip compressed writing.</td>
</tr>
<tr>
<td>'w:bz2'</td>
<td>Open for bzip2 compressed writing.</td>
</tr>
</tbody>
</table>

Note that 'a:gz' or 'a:bz2' is not possible. If mode is not suitable to open a certain (compressed) file for reading, ReadError is raised. Use mode 'r' to avoid this. If a compression method is not supported, CompressionError is raised.

If fileobj is specified, it is used as an alternative to a file object opened for name. It is supposed to be at position 0.

For modes 'w:gz', 'r:gz', 'w:bz2', 'r:bz2', tarfile.open() accepts the keyword argument compresslevel (default 9) to specify the compression level of the file.

For special purposes, there is a second format for mode: 'filemode[:compression]'. tarfile.open() will return a TarFile
object that processes its data as a stream of blocks. No random seeking will be done on the file. If given, fileobj may be any object that has a read() or write() method (depending on the mode). bufsize specifies the blocksize and defaults to 20 * 512 bytes. Use this variant in combination with e.g. sys.stdin, a socket file object or a tape device. However, such a TarFile object is limited in that it does not allow random access, see Examples. The currently possible modes:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r</td>
<td>*'</td>
</tr>
<tr>
<td>'r</td>
<td></td>
</tr>
<tr>
<td>'r</td>
<td>gz'</td>
</tr>
<tr>
<td>'r</td>
<td>bz2'</td>
</tr>
<tr>
<td>'w</td>
<td></td>
</tr>
<tr>
<td>'w</td>
<td>gz'</td>
</tr>
<tr>
<td>'w</td>
<td>bz2'</td>
</tr>
</tbody>
</table>

class tarfile.TarFile

Class for reading and writing tar archives. Do not use this class directly, better use tarfile.open() instead. See TarFile Objects.

tarfile.is_tarfile(name)

Return True if name is a tar archive file, that the tarfile module can read.

class tarfile.TarFileCompat(filename, mode='r', compression=TAR_PLAIN)

Class for limited access to tar archives with a zipfile-like interface. Please consult the documentation of the zipfile module for more details. compression must be one of the following constants:
**TAR_PLAIN**
Constant for an uncompressed tar archive.

**TAR_GZIPPED**
Constant for a gzip compressed tar archive.

*Deprecated since version 2.6:* The `TarFileCompat` class has been removed in Python 3.

*exception* `tarfile.TarError`
Base class for all `tarfile` exceptions.

*exception* `tarfile.ReadError`
Is raised when a tar archive is opened, that either cannot be handled by the `tarfile` module or is somehow invalid.

*exception* `tarfile.CompressionError`
Is raised when a compression method is not supported or when the data cannot be decoded properly.

*exception* `tarfile.StreamError`
Is raised for the limitations that are typical for stream-like `TarFile` objects.

*exception* `tarfile.ExtractError`
Is raised for *non-fatal* errors when using `TarFile.extract()`, but only if `TarFile.errorlevel==2`.

The following constants are available at the module level:

*tarfile.ENCODING*
The default character encoding: 'utf-8' on Windows, the value returned by `sys.getfilesystemencoding()` otherwise.

*exception* `tarfile.HeaderError`
Is raised by `TarInfo.frombuf()` if the buffer it gets is invalid.

*New in version 2.6.*

Each of the following constants defines a tar archive format that the `tarfile` module is able to create. See section `Supported tar formats` for details.

- `tarfile.USTAR_FORMAT`
  POSIX.1-1988 (ustar) format.

- `tarfile.GNU_FORMAT`
  GNU tar format.

- `tarfile.PAX_FORMAT`
  POSIX.1-2001 (pax) format.

- `tarfile.DEFAULT_FORMAT`
  The default format for creating archives. This is currently `GNU_FORMAT`.

**See also:**

- **Module `zipfile`**
  Documentation of the `zipfile` standard module.

- **Archiving operations**
  Documentation of the higher-level archiving facilities provided by the standard `shutil` module.

- **GNU tar manual, Basic Tar Format**
  Documentation for tar archive files, including GNU tar extensions.
12.5.1. TarFile Objects

The **TarFile** object provides an interface to a tar archive. A tar archive is a sequence of blocks. An archive member (a stored file) is made up of a header block followed by data blocks. It is possible to store a file in a tar archive several times. Each archive member is represented by a **TarInfo** object, see **TarInfo Objects** for details.

A **TarFile** object can be used as a context manager in a `with` statement. It will automatically be closed when the block is completed. Please note that in the event of an exception an archive opened for writing will not be finalized; only the internally used file object will be closed. See the **Examples** section for a use case.

New in version 2.7: Added support for the context management protocol.

```python
class tarfile.TarFile(name=None, mode='r', fileobj=None,
                      format=DEFAULT_FORMAT, tarinfo=TarInfo, dereference=False,
                      ignore_zeros=False, encoding=ENCODING, errors=None,
                      pax_headers=None, debug=0, errorlevel=0)
```

All following arguments are optional and can be accessed as instance attributes as well.

- **name** is the pathname of the archive. It can be omitted if `fileobj` is given. In this case, the file object’s `name` attribute is used if it exists.

- **mode** is either `'r'` to read from an existing archive, `'a'` to append data to an existing file or `'w'` to create a new file overwriting an existing one.

If `fileobj` is given, it is used for reading or writing data. If it can be determined, `mode` is overridden by `fileobj`'s mode. `fileobj` will be
used from position 0.

**Note:** fileobj is not closed, when TarFile is closed.

format controls the archive format. It must be one of the constants USTAR_FORMAT, GNU_FORMAT or PAX_FORMAT that are defined at module level.

*New in version 2.6.*

The tarinfo argument can be used to replace the default TarInfo class with a different one.

*New in version 2.6.*

If dereference is False, add symbolic and hard links to the archive. If it is True, add the content of the target files to the archive. This has no effect on systems that do not support symbolic links.

If ignore_zeros is False, treat an empty block as the end of the archive. If it is True, skip empty (and invalid) blocks and try to get as many members as possible. This is only useful for reading concatenated or damaged archives.

debug can be set from 0 (no debug messages) up to 3 (all debug messages). The messages are written to sys.stderr.

If errorlevel is 0, all errors are ignored when using TarFile.extract(). Nevertheless, they appear as error messages in the debug output, when debugging is enabled. If 1, all fatal errors are raised as OSError or IOError exceptions. If 2, all non-fatal errors are raised as TarError exceptions as well.

The encoding and errors arguments control the way strings are
converted to unicode objects and vice versa. The default settings will work for most users. See section Unicode issues for in-depth information.

*New in version 2.6.*

The `pax_headers` argument is an optional dictionary of unicode strings which will be added as a pax global header if `format` is `PAX_FORMAT`.

*New in version 2.6.*

classmethod `TarFile.open(...)`

Alternative constructor. The `tarfile.open()` function is actually a shortcut to this classmethod.

`TarFile.getmember(name)`

Return a `TarInfo` object for member `name`. If `name` can not be found in the archive, `KeyError` is raised.

**Note:** If a member occurs more than once in the archive, its last occurrence is assumed to be the most up-to-date version.

`TarFile.getmembers()`

Return the members of the archive as a list of `TarInfo` objects. The list has the same order as the members in the archive.

`TarFile.getnames()`

Return the members as a list of their names. It has the same order as the list returned by `getmembers()`.

`TarFile.list(verbosity=True)`

Print a table of contents to `sys.stdout`. If `verbose` is `False`, only the names of the members are printed. If it is `True`, output similar
to that of `ls -l` is produced.

`TarFile.next()`

Return the next member of the archive as a `TarInfo` object, when `TarFile` is opened for reading. Return `None` if there is no more available.

`TarFile.extractall(path=".", members=None)`

Extract all members from the archive to the current working directory or directory `path`. If optional `members` is given, it must be a subset of the list returned by `getmembers()`. Directory information like owner, modification time and permissions are set after all members have been extracted. This is done to work around two problems: A directory's modification time is reset each time a file is created in it. And, if a directory's permissions do not allow writing, extracting files to it will fail.

**Warning:** Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of `path`, e.g. members that have absolute filenames starting with `"/"` or filenames with two dots `".."`.

*New in version 2.5.*

`TarFile.extract(member, path="")`

Extract a member from the archive to the current working directory, using its full name. Its file information is extracted as accurately as possible. `member` may be a filename or a `TarInfo` object. You can specify a different directory using `path`.

**Note:** The `extract()` method does not take care of several extraction issues. In most cases you should consider using the `extractall()` method.
Warning: See the warning for extractall().

**TarFile**. `extractfile(member)`

Extract a member from the archive as a file object. `member` may be a filename or a **TarInfo** object. If `member` is a regular file, a file-like object is returned. If `member` is a link, a file-like object is constructed from the link's target. If `member` is none of the above, **None** is returned.

**Note**: The file-like object is read-only. It provides the methods `read()`, `readline()`, `readlines()`, `seek()`, `tell()`, and `close()`, and also supports iteration over its lines.

**TarFile**. `add(name, arcname=None, recursive=True, exclude=None, filter=None)`

Add the file `name` to the archive. `name` may be any type of file (directory, fifo, symbolic link, etc.). If given, `arcname` specifies an alternative name for the file in the archive. Directories are added recursively by default. This can be avoided by setting `recursive` to **False**. If `exclude` is given it must be a function that takes one filename argument and returns a boolean value. Depending on this value the respective file is either excluded (**True**) or added (**False**). If `filter` is specified it must be a function that takes a **TarInfo** object argument and returns the changed **TarInfo** object. If it instead returns **None** the **TarInfo** object will be excluded from the archive. See **Examples** for an example.

*Changed in version 2.6*: Added the `exclude` parameter.

*Changed in version 2.7*: Added the `filter` parameter.

*Deprecated since version 2.7*: The `exclude` parameter is deprecated, please use the `filter` parameter instead. For
maximum portability, *filter* should be used as a keyword argument rather than as a positional argument so that code won’t be affected when *exclude* is ultimately removed.

**TarFile.** `addfile(tarinfo, fileobj=None)`

Add the *TarInfo* object `tarinfo` to the archive. If `fileobj` is given, `tarinfo.size` bytes are read from it and added to the archive. You can create *TarInfo* objects directly, or by using `gettarinfo()`.

**Note:** On Windows platforms, `fileobj` should always be opened with mode `'rb'` to avoid irritation about the file size.

**TarFile.** `gettarinfo(name=None, arcname=None, fileobj=None)`

Create a *TarInfo* object from the result of `os.stat()` or equivalent on an existing file. The file is either named by `name`, or specified as a file object `fileobj` with a file descriptor. If given, `arcname` specifies an alternative name for the file in the archive, otherwise, the name is taken from `fileobj`'s `name` attribute, or the `name` argument.

You can modify some of the *TarInfo*’s attributes before you add it using `addfile()`. If the file object is not an ordinary file object positioned at the beginning of the file, attributes such as `size` may need modifying. This is the case for objects such as `GzipFile`. The `name` may also be modified, in which case `arcname` could be a dummy string.

**TarFile.** `close()`

Close the *TarFile*. In write mode, two finishing zero blocks are appended to the archive.

**TarFile.** `posix`
Setting this to `True` is equivalent to setting the `format` attribute to `USTAR_FORMAT`, `False` is equivalent to `GNU_FORMAT`.

*Changed in version 2.4:* `posix` defaults to `False`.

*Deprecated since version 2.6:* Use the `format` attribute instead.

**TarFile.** `pax_headers`

A dictionary containing key-value pairs of pax global headers.

*New in version 2.6.*
12.5.2. TarInfo Objects

A TarInfo object represents one member in a TarFile. Aside from storing all required attributes of a file (like file type, size, time, permissions, owner etc.), it provides some useful methods to determine its type. It does not contain the file's data itself.

TarInfo objects are returned by TarFile's methods getmember(), getmembers() and gettarinfo().

```python
class tarfile.TarInfo(name="")
    Create a TarInfo object.

TarInfo.frombuf(buf)
    Create and return a TarInfo object from string buffer buf.

    New in version 2.6: Raises HeaderError if the buffer is invalid.

TarInfo.fromtarfile(tarfile)
    Read the next member from the TarFile object tarfile and return it as a TarInfo object.

    New in version 2.6.

TarInfo.tobuf(format=DEFAULT_FORMAT, encoding=ENCODING, errors='strict')
    Create a string buffer from a TarInfo object. For information on the arguments see the constructor of the TarFile class.

    Changed in version 2.6: The arguments were added.
```

A TarInfo object has the following public data attributes:

```
TarInfo.name
```
Name of the archive member.

**`TarInfo.size`**
Size in bytes.

**`TarInfo.mtime`**
Time of last modification.

**`TarInfo.mode`**
Permission bits.

**`TarInfo.type`**
File type. `type` is usually one of these constants: `REGTYPE`, `AREGTYPE`, `LNKTYPE`, `SYMTYPE`, `DIRTYPE`, `FIFOTYPE`, `CONTTYPE`, `CHRTYPE`, `BLKTYPE`, `GNUTYPE_SPARSE`. To determine the type of a `TarInfo` object more conveniently, use the `is*()` methods below.

**`TarInfo.linkname`**
Name of the target file name, which is only present in `TarInfo` objects of type `LNKTYPE` and `SYMTYPE`.

**`TarInfo.uid`**
User ID of the user who originally stored this member.

**`TarInfo.gid`**
Group ID of the user who originally stored this member.

**`TarInfo.uname`**
User name.

**`TarInfo.gname`**
Group name.

**`TarInfo.pax_headers`**
A dictionary containing key-value pairs of an associated pax extended header.
New in version 2.6.

A `TarInfo` object also provides some convenient query methods:

`TarInfo.isfile()`  
Return `True` if the `TarInfo` object is a regular file.

`TarInfo.isreg()`  
Same as `isfile()`.

`TarInfo.isdir()`  
Return `True` if it is a directory.

`TarInfo.issym()`  
Return `True` if it is a symbolic link.

`TarInfo.islnk()`  
Return `True` if it is a hard link.

`TarInfo.ischr()`  
Return `True` if it is a character device.

`TarInfo.isblk()`  
Return `True` if it is a block device.

`TarInfo.isfifo()`  
Return `True` if it is a FIFO.

`TarInfo.isdev()`  
Return `True` if it is one of character device, block device or FIFO.
12.5.3. Examples

How to extract an entire tar archive to the current working directory:

```python
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

How to extract a subset of a tar archive with `TarFile.extractall()` using a generator function instead of a list:

```python
import os
import tarfile
def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

How to create an uncompressed tar archive from a list of filenames:

```python
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:  
    tar.add(name)
tar.close()
```

The same example using the `with` statement:

```python
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:  
        tar.add(name)
```
How to read a gzip compressed tar archive and display some member information:

```python
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print tarinfo.name, "is", tarinfo.size, "bytes in size and"
    if tarinfo.isfile():
        print "a regular file."
    elif tarinfo.isdir():
        print "a directory."
    else:
        print "something else."
tar.close()
```

How to create an archive and reset the user information using the `filter` parameter in `TarFile.add()`:

```python
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```
12.5.4. Supported tar formats

There are three tar formats that can be created with the `tarfile` module:

- The POSIX.1-1988 ustar format (**USTAR_FORMAT**). It supports filenames up to a length of at best 256 characters and linknames up to 100 characters. The maximum file size is 8 gigabytes. This is an old and limited but widely supported format.

- The GNU tar format (**GNU_FORMAT**). It supports long filenames and linknames, files bigger than 8 gigabytes and sparse files. It is the de facto standard on GNU/Linux systems. `tarfile` fully supports the GNU tar extensions for long names, sparse file support is read-only.

- The POSIX.1-2001 pax format (**PAX_FORMAT**). It is the most flexible format with virtually no limits. It supports long filenames and linknames, large files and stores pathnames in a portable way. However, not all tar implementations today are able to handle pax archives properly.

  The *pax* format is an extension to the existing *ustar* format. It uses extra headers for information that cannot be stored otherwise. There are two flavours of pax headers: Extended headers only affect the subsequent file header, global headers are valid for the complete archive and affect all following files. All the data in a pax header is encoded in UTF-8 for portability reasons.

  There are some more variants of the tar format which can be read, but not created:
- The ancient V7 format. This is the first tar format from Unix Seventh Edition, storing only regular files and directories. Names must not be longer than 100 characters, there is no user/group name information. Some archives have miscalculated header checksums in case of fields with non-ASCII characters.
- The SunOS tar extended format. This format is a variant of the POSIX.1-2001 pax format, but is not compatible.
12.5.5. Unicode issues

The tar format was originally conceived to make backups on tape drives with the main focus on preserving file system information. Nowadays tar archives are commonly used for file distribution and exchanging archives over networks. One problem of the original format (that all other formats are merely variants of) is that there is no concept of supporting different character encodings. For example, an ordinary tar archive created on a UTF-8 system cannot be read correctly on a Latin-1 system if it contains non-ASCII characters. Names (i.e. filenames, linknames, user/group names) containing these characters will appear damaged. Unfortunately, there is no way to autodetect the encoding of an archive.

The pax format was designed to solve this problem. It stores non-ASCII names using the universal character encoding UTF-8. When a pax archive is read, these UTF-8 names are converted to the encoding of the local file system.

The details of unicode conversion are controlled by the *encoding* and *errors* keyword arguments of the *TarFile* class.

The default value for *encoding* is the local character encoding. It is deduced from *sys.getfilesystemencoding()* and *sys.getdefaultencoding()* in read mode, *encoding* is used exclusively to convert unicode names from a pax archive to strings in the local character encoding. In write mode, the use of *encoding* depends on the chosen archive format. In case of PAX_FORMAT, input names that contain non-ASCII characters need to be decoded before being stored as UTF-8 strings. The other formats do not make use of *encoding* unless unicode objects are used as input names. These are converted to 8-bit character strings before they are added to the archive.
The errors argument defines how characters are treated that cannot be converted to or from encoding. Possible values are listed in section Codec Base Classes. In read mode, there is an additional scheme 'utf-8' which means that bad characters are replaced by their UTF-8 representation. This is the default scheme. In write mode the default value for errors is 'strict' to ensure that name information is not altered unnoticed.
13. File Formats

The modules described in this chapter parse various miscellaneous file formats that aren’t markup languages or are related to e-mail.

- **13.1. csv** — CSV File Reading and Writing
  - 13.1.1. Module Contents
  - 13.1.2. Dialects and Formatting Parameters
  - 13.1.3. Reader Objects
  - 13.1.4. Writer Objects
  - 13.1.5. Examples

- **13.2. ConfigParser** — Configuration file parser
  - 13.2.1. RawConfigParser Objects
  - 13.2.2. ConfigParser Objects
  - 13.2.3. SafeConfigParser Objects
  - 13.2.4. Examples

- **13.3. robotparser** — Parser for robots.txt

- **13.4. netrc** — netrc file processing
  - 13.4.1. netrc Objects

- **13.5. xdrlib** — Encode and decode XDR data
  - 13.5.1. Packer Objects
  - 13.5.2. Unpacker Objects
  - 13.5.3. Exceptions

- **13.6. plistlib** — Generate and parse Mac OS X .plist files
  - 13.6.1. Examples
13.1. **csv** — CSV File Reading and Writing

*New in version 2.3.*

The so-called CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. There is no “CSV standard”, so the format is operationally defined by the many applications which read and write it. The lack of a standard means that subtle differences often exist in the data produced and consumed by different applications. These differences can make it annoying to process CSV files from multiple sources. Still, while the delimiters and quoting characters vary, the overall format is similar enough that it is possible to write a single module which can efficiently manipulate such data, hiding the details of reading and writing the data from the programmer.

The **csv** module implements classes to read and write tabular data in CSV format. It allows programmers to say, “write this data in the format preferred by Excel,” or “read data from this file which was generated by Excel,” without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.

The **csv** module’s **reader** and **writer** objects read and write sequences. Programmers can also read and write data in dictionary form using the **DictReader** and **DictWriter** classes.

**Note:** This version of the **csv** module doesn’t support Unicode input. Also, there are currently some issues regarding ASCII NUL characters. Accordingly, all input should be UTF-8 or printable ASCII to be safe; see the examples in section **Examples**.
See also:

**PEP 305 - CSV File API**

The Python Enhancement Proposal which proposed this addition to Python.
13.1.1. Module Contents

The csv module defines the following functions:

```
csv.reader(csvfile, dialect='excel', **fmtparams)
```

Return a reader object which will iterate over lines in the given `csvfile`. `csvfile` can be any object which supports the iterator protocol and returns a string each time its `next()` method is called — file objects and list objects are both suitable. If `csvfile` is a file object, it must be opened with the ‘b’ flag on platforms where that makes a difference. An optional `dialect` parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the `Dialect` class or one of the strings returned by the `list_dialects()` function. The other optional `fmtparams` keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters.

Each row read from the csv file is returned as a list of strings. No automatic data type conversion is performed.

A short usage example:

```
>>> import csv
>>> with open('eggs.csv', 'rb') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print ', '.join(row)
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

*Changed in version 2.5:* The parser is now stricter with respect to
multi-line quoted fields. Previously, if a line ended within a quoted field without a terminating newline character, a newline would be inserted into the returned field. This behavior caused problems when reading files which contained carriage return characters within fields. The behavior was changed to return the field without inserting newlines. As a consequence, if newlines embedded within fields are important, the input should be split into lines in a manner which preserves the newline characters.

csv.writer(csvfile, dialect='excel', **fmtparams)

Return a writer object responsible for converting the user’s data into delimited strings on the given file-like object. csvfile can be any object with a write() method. If csvfile is a file object, it must be opened with the ‘b’ flag on platforms where that makes a difference. An optional dialect parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the Dialect class or one of the strings returned by the list_dialects() function. The other optional fmtparams keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters. To make it as easy as possible to interface with modules which implement the DB API, the value None is written as the empty string. While this isn’t a reversible transformation, it makes it easier to dump SQL NULL data values to CSV files without preprocessing the data returned from a cursor.fetchall call. Floats are stringified with repr() before being written. All other non-string data are stringified with str() before being written.

A short usage example:

```python
import csv
with open('eggs.csv', 'wb') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
```
csv.register_dialect(name, [dialect, ]**fmtparams)

Associate dialect with name. name must be a string or Unicode object. The dialect can be specified either by passing a sub-class of Dialect, or by fmtparams keyword arguments, or both, with keyword arguments overriding parameters of the dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters.

csv.unregister_dialect(name)

Delete the dialect associated with name from the dialect registry. An Error is raised if name is not a registered dialect name.

csv.get_dialect(name)

Return the dialect associated with name. An Error is raised if name is not a registered dialect name.

Changed in version 2.5: This function now returns an immutable Dialect. Previously an instance of the requested dialect was returned. Users could modify the underlying class, changing the behavior of active readers and writers.

csv.list_dialects()

Return the names of all registered dialects.

csv.field_size_limit([new_limit])

Returns the current maximum field size allowed by the parser. If new_limit is given, this becomes the new limit.

New in version 2.5.
The `csv` module defines the following classes:

```python
class csv.DictReader(csvfile, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kwds)
```

Create an object which operates like a regular reader but maps the information read into a dict whose keys are given by the optional `fieldnames` parameter. The `fieldnames` parameter is a sequence whose elements are associated with the fields of the input data in order. These elements become the keys of the resulting dictionary. If the `fieldnames` parameter is omitted, the values in the first row of the `csvfile` will be used as the fieldnames. If the row read has more fields than the fieldnames sequence, the remaining data is added as a sequence keyed by the value of `restkey`. If the row read has fewer fields than the fieldnames sequence, the remaining keys take the value of the optional `restval` parameter. Any other optional or keyword arguments are passed to the underlying `reader` instance.

A short usage example:

```python
>>> import csv
>>> with open('names.csv') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...     Baked Beans
...     Lovely Spam
...     Wonderful Spam
```

```python
class csv.DictWriter(csvfile, fieldnames, restval='', extrasaction='raise', dialect='excel', *args, **kwds)
```

Create an object which operates like a regular writer but maps dictionaries onto output rows. The `fieldnames` parameter is a sequence of keys that identify the order in which values in the dictionary passed to the `writerow()` method are written to the `csvfile`. The optional `restval` parameter specifies the value to be
written if the dictionary is missing a key in fieldnames. If the
dictionary passed to the writerow() method contains a key not
found in fieldnames, the optional extrasaction parameter
indicates what action to take. If it is set to 'raise' a ValueError
is raised. If it is set to 'ignore', extra values in the dictionary are
ignored. Any other optional or keyword arguments are passed to
the underlying writer instance.

Note that unlike the DictReader class, the fieldnames parameter
of the DictWriter is not optional. Since Python’s dict objects are
not ordered, there is not enough information available to deduce
the order in which the row should be written to the csvfile.

A short usage example:

```python
import csv

with open('names.csv', 'w') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

class csv.Dialect

The Dialect class is a container class relied on primarily for its
attributes, which are used to define the parameters for a specific
reader or writer instance.

class csv.excel

The excel class defines the usual properties of an Excel-
generated CSV file. It is registered with the dialect name 'excel'.

class csv.excel_tab
The `excel_tab` class defines the usual properties of an Excel-generated TAB-delimited file. It is registered with the dialect name `'excel-tab'`.

```python
class csv.Sniffer
    The Sniffer class is used to deduce the format of a CSV file.

    The Sniffer class provides two methods:

    `sniff(sample, delimiters=None)`
    Analyze the given `sample` and return a `Dialect` subclass reflecting the parameters found. If the optional `delimiters` parameter is given, it is interpreted as a string containing possible valid delimiter characters.

    `has_header(sample)`
    Analyze the sample text (presumed to be in CSV format) and return `True` if the first row appears to be a series of column headers.

An example for Sniffer use:

```python
with open('example.csv', 'rb') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

The `csv` module defines the following constants:

```python
csv.QUOTE_ALL
    Instructs `writer` objects to quote all fields.

csv.QUOTE_MINIMAL
    Instructs `writer` objects to only quote those fields which contain special characters such as `delimiter`, `quotechar` or any of the
characters in \textit{lineterminator}.

\textbf{csv. \texttt{QUOTE\_NONNUMERIC}}

Instructs \texttt{writer} objects to quote all non-numeric fields.

Instructs the reader to convert all non-quoted fields to type \textit{float}.

\textbf{csv. \texttt{QUOTE\_NONE}}

Instructs \texttt{writer} objects to never quote fields. When the current \texttt{delimiter} occurs in output data it is preceded by the current \texttt{escapechar} character. If \texttt{escapechar} is not set, the writer will raise \texttt{Error} if any characters that require escaping are encountered.

Instructs \texttt{reader} to perform no special processing of quote characters.

The \texttt{csv} module defines the following exception:

\textbf{exception \texttt{csv.\texttt{Error}}}

Raised by any of the functions when an error is detected.
13.1.2. Dialects and Formatting Parameters

To make it easier to specify the format of input and output records, specific formatting parameters are grouped together into dialects. A dialect is a subclass of the \texttt{Dialect} class having a set of specific methods and a single \texttt{validate()} method. When creating \texttt{reader} or \texttt{writer} objects, the programmer can specify a string or a subclass of the \texttt{Dialect} class as the dialect parameter. In addition to, or instead of, the \texttt{dialect} parameter, the programmer can also specify individual formatting parameters, which have the same names as the attributes defined below for the \texttt{Dialect} class.

Dialects support the following attributes:

\texttt{Dialect.delimiter}

A one-character string used to separate fields. It defaults to ",".

\texttt{Dialect.doublequote}

Controls how instances of \texttt{quotechar} appearing inside a field should themselves be quoted. When \texttt{True}, the character is doubled. When \texttt{False}, the \texttt{escapechar} is used as a prefix to the \texttt{quotechar}. It defaults to \texttt{True}.

On output, if \texttt{doublequote} is \texttt{False} and no \texttt{escapechar} is set, \texttt{Error} is raised if a \texttt{quotechar} is found in a field.

\texttt{Dialect.escapechar}

A one-character string used by the writer to escape the \texttt{delimiter} if \texttt{quoting} is set to \texttt{QUOTE_NONE} and the \texttt{quotechar} if \texttt{doublequote} is \texttt{False}. On reading, the \texttt{escapechar} removes any special meaning from the following character. It defaults to \texttt{None}, which disables escaping.
Dialect. **lineterminator**

The string used to terminate lines produced by the writer. It defaults to '\r\n'.

**Note:** The reader is hard-coded to recognise either '\r' or '\n' as end-of-line, and ignores *lineterminator*. This behavior may change in the future.

Dialect. **quotechar**

A one-character string used to quote fields containing special characters, such as the *delimiter* or *quotechar*, or which contain new-line characters. It defaults to '"'.

Dialect. **quoting**

Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the *QUOTE_* constants (see section Module Contents) and defaults to *QUOTE_MINIMAL*.

Dialect. **skipinitialspace**

When *True*, whitespace immediately following the *delimiter* is ignored. The default is *False*.

Dialect. **strict**

When *True*, raise exception *Error* on bad CSV input. The default is *False*. 
13.1.3. Reader Objects

Reader objects (DictReader instances and objects returned by the reader() function) have the following public methods:

csvreader.next()

    Return the next row of the reader's iterable object as a list, parsed according to the current dialect.

Reader objects have the following public attributes:

csvreader.dialect

    A read-only description of the dialect in use by the parser.

csvreader.line_num

    The number of lines read from the source iterator. This is not the same as the number of records returned, as records can span multiple lines.

    New in version 2.5.

DictReader objects have the following public attribute:

csvreader.fieldnames

    If not passed as a parameter when creating the object, this attribute is initialized upon first access or when the first record is read from the file.

    Changed in version 2.6.
13.1.4. Writer Objects

`Writer` objects (`DictWriter` instances and objects returned by the `writer()` function) have the following public methods. A row must be a sequence of strings or numbers for `Writer` objects and a dictionary mapping fieldnames to strings or numbers (by passing them through `str()` first) for `DictWriter` objects. Note that complex numbers are written out surrounded by parens. This may cause some problems for other programs which read CSV files (assuming they support complex numbers at all).

```
csvwriter.writerow(row)
```

Write the `row` parameter to the writer’s file object, formatted according to the current dialect.

```
csvwriter.writerows(rows)
```

Write all the `rows` parameters (a list of `row` objects as described above) to the writer’s file object, formatted according to the current dialect.

Writer objects have the following public attribute:

```
csvwriter.dialect
```

A read-only description of the dialect in use by the writer.

DictWriter objects have the following public method:

```
DictWriter.writerow()
```

Write a row with the field names (as specified in the constructor).

*New in version 2.7.*
13.1.5. Examples

The simplest example of reading a CSV file:

```
import csv
with open('some.csv', 'rb') as f:
    reader = csv.reader(f)
    for row in reader:
        print row
```

Reading a file with an alternate format:

```
import csv
with open('passwd', 'rb') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print row
```

The corresponding simplest possible writing example is:

```
import csv
with open('some.csv', 'wb') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

Registering a new dialect:

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', 'rb') as f:
    reader = csv.reader(f, 'unixpwd')
```

A slightly more advanced use of the reader — catching and reporting errors:

```
import csv, sys
filename = 'some.csv'
```
And while the module doesn’t directly support parsing strings, it can easily be done:

```python
import csv
for row in csv.reader(['one,two,three']):
    print row
```

The `csv` module doesn’t directly support reading and writing Unicode, but it is 8-bit-clean save for some problems with ASCII NUL characters. So you can write functions or classes that handle the encoding and decoding for you as long as you avoid encodings like UTF-16 that use NULs. UTF-8 is recommended.

`unicode_csv_reader()` below is a generator that wraps `csv.reader` to handle Unicode CSV data (a list of Unicode strings). `utf_8_encoder()` is a generator that encodes the Unicode strings as UTF-8, one string (or row) at a time. The encoded strings are parsed by the CSV reader, and `unicode_csv_reader()` decodes the UTF-8-encoded cells back into Unicode:

```python
import csv

def unicode_csv_reader(unicode_csv_data, dialect=csv.excel, **kwargs):
    # csv.py doesn't do Unicode; encode temporarily as UTF-8:
    csv_reader = csv.reader(utf_8_encoder(unicode_csv_data),
                            dialect=dialect, **kwargs)
    for row in csv_reader:
        # decode UTF-8 back to Unicode, cell by cell:
        yield [unicode(cell, 'utf-8') for cell in row]

def utf_8_encoder(unicode_csv_data):
```

```
for line in unicode_csv_data:
    yield line.encode('utf-8')

For all other encodings the following **UnicodeReader** and **UnicodeWriter** classes can be used. They take an additional *encoding* parameter in their constructor and make sure that the data passes the real reader or writer encoded as UTF-8:

```python
import csv, codecs, cStringIO

class UTF8Recoder:
    """
    Iterator that reads an encoded stream and reencodes the input
    to UTF-8.
    """
    def __init__(self, f, encoding):
        self.reader = codecs.getreader(encoding)(f)

    def __iter__(self):
        return self

    def next(self):
        return self.reader.next().encode("utf-8")

class UnicodeReader:
    """
    A CSV reader which will iterate over lines in the CSV file which is encoded in the given encoding.
    """
    def __init__(self, f, dialect=csv.excel, encoding="utf-8", **kwds):
        f = UTF8Recoder(f, encoding)
        self.reader = csv.reader(f, dialect=dialect, **kwds)

    def next(self):
        row = self.reader.next()
        return [unicode(s, "utf-8") for s in row]

    def __iter__(self):
        return self

class UnicodeWriter:
    """
    A CSV writer which will write rows to CSV file "f", which is encoded in the given encoding.
    """
```
```python
def __init__(self, f, dialect=csv.xlsx, encoding="utf-8", 
    # Redirect output to a queue
    self.queue = cStringIO.StringIO()
    self.writer = csv.writer(self.queue, dialect=dialect, 
    self.stream = f
    self.encoder = codecs.getincrementalencoder(encoding)()

def writerow(self, row):
    self.writer.writerow([s.encode("utf-8") for s in row])
    # Fetch UTF-8 output from the queue ...
    data = self.queue.getvalue()
    data = data.decode("utf-8")
    # ... and reencode it into the target encoding
    data = self.encoder.encode(data)
    # write to the target stream
    self.stream.write(data)
    # empty queue
    self.queue.truncate(0)

def writerows(self, rows):
    for row in rows:
        self.writerow(row)
```

13.2. ConfigParser — Configuration file parser

**Note:** The ConfigParser module has been renamed to configparser in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

This module defines the class ConfigParser. The ConfigParser class implements a basic configuration file parser language which provides a structure similar to what you would find on Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

**Note:** This library does not interpret or write the value-type prefixes used in the Windows Registry extended version of INI syntax.

**See also:**

**Module shlex**

Support for creating Unix shell-like mini-languages which can be used as an alternate format for application configuration files.

**Module json**

The json module implements a subset of JavaScript syntax which can also be used for this purpose.

The configuration file consists of sections, led by a [section] header and followed by name: value entries, with continuations in the style of RFC 822 (see section 3.1.1, “LONG HEADER FIELDS”); name=value is also accepted. Note that leading whitespace is
removed from values. The optional values can contain format strings which refer to other values in the same section, or values in a special DEFAULT section. Additional defaults can be provided on initialization and retrieval. Lines beginning with '#' or ';' are ignored and may be used to provide comments.

Configuration files may include comments, prefixed by specific characters (# and ;). Comments may appear on their own in an otherwise empty line, or may be entered in lines holding values or section names. In the latter case, they need to be preceded by a whitespace character to be recognized as a comment. (For backwards compatibility, only ; starts an inline comment, while # does not.)

On top of the core functionality, SafeConfigParser supports interpolation. This means values can contain format strings which refer to other values in the same section, or values in a special DEFAULT section. Additional defaults can be provided on initialization.

For example:

```plaintext
[My Section]
foodir: %(dir)s/whatever
dir=frob
long: this value continues
     in the next line
```

would resolve the %(dir)s to the value of dir (frob in this case). All reference expansions are done on demand.

Default values can be specified by passing them into the ConfigParser constructor as a dictionary. Additional defaults may be passed into the get() method which will override all others.

Sections are normally stored in a built-in dictionary. An alternative
dictionary type can be passed to the `ConfigParser` constructor. For example, if a dictionary type is passed that sorts its keys, the sections will be sorted on write-back, as will be the keys within each section.

```python
class ConfigParser. RawConfigParser( [defaults, dict_type, allow_no_value]])
```

The basic configuration object. When `defaults` is given, it is initialized into the dictionary of intrinsic defaults. When `dict_type` is given, it will be used to create the dictionary objects for the list of sections, for the options within a section, and for the default values. When `allow_no_value` is true (default: `False`), options without values are accepted; the value presented for these is `None`.

This class does not support the magical interpolation behavior.

All option names are passed through the `optionxform()` method. Its default implementation converts option names to lower case.

*New in version 2.3.*

*Changed in version 2.6:* `dict_type` was added.

*Changed in version 2.7:* The default `dict_type` is `collections.OrderedDict`. `allow_no_value` was added.

```python
class ConfigParser. ConfigParser( [defaults, dict_type, allow_no_value]])
```

Derived class of `RawConfigParser` that implements the magical interpolation feature and adds optional arguments to the `get()` and `items()` methods. The values in `defaults` must be appropriate for the `%()s` string interpolation. Note that `__name__` is an
intrinsic default; its value is the section name, and will override any value provided in defaults.

All option names used in interpolation will be passed through the optionxform() method just like any other option name reference. Using the default implementation of optionxform(), the values foo %(bar)s and foo %(BAR)s are equivalent.

*New in version 2.3.*

*Changed in version 2.6:* dict_type was added.

*Changed in version 2.7:* The default dict_type is collections.OrderedDict. allow_no_value was added.

```python
class ConfigParser. SafeConfigParser([defaults[, dict_type[, allow_no_value]]])
```

Derived class of ConfigParser that implements a more-sane variant of the magical interpolation feature. This implementation is more predictable as well. New applications should prefer this version if they don’t need to be compatible with older versions of Python.

*New in version 2.3.*

*Changed in version 2.6:* dict_type was added.

*Changed in version 2.7:* The default dict_type is collections.OrderedDict. allow_no_value was added.

```python
exception ConfigParser. Error
```

Base class for all other configparser exceptions.

```python
exception ConfigParser. NoSectionError
```

Exception raised when a specified section is not found.
exception ConfigParser.DuplicateSectionError
   Exception raised if add_section() is called with the name of a section that is already present.

exception ConfigParser.NoOptionError
   Exception raised when a specified option is not found in the specified section.

exception ConfigParser.InterpolationError
   Base class for exceptions raised when problems occur performing string interpolation.

exception ConfigParser.InterpolationDepthError
   Exception raised when string interpolation cannot be completed because the number of iterations exceeds MAX_INTERPOLATION_DEPTH. Subclass of InterpolationError.

exception ConfigParser.InterpolationMissingOptionError
   Exception raised when an option referenced from a value does not exist. Subclass of InterpolationError.

   New in version 2.3.

exception ConfigParser.InterpolationSyntaxError
   Exception raised when the source text into which substitutions are made does not conform to the required syntax. Subclass of InterpolationError.

   New in version 2.3.

exception ConfigParser.MissingSectionHeaderError
   Exception raised when attempting to parse a file which has no section headers.

exception ConfigParser.ParsingError
Exception raised when errors occur attempting to parse a file.

`ConfigParser.MAX_INTERPOLATION_DEPTH`

The maximum depth for recursive interpolation for `get()` when the `raw` parameter is false. This is relevant only for the `ConfigParser` class.

See also:

**Module `shlex`**

Support for a creating Unix shell-like mini-languages which can be used as an alternate format for application configuration files.
13.2.1. RawConfigParser Objects

RawConfigParser instances have the following methods:

RawConfigParser.**defaults()**
Return a dictionary containing the instance-wide defaults.

RawConfigParser.**sections()**
Return a list of the sections available; DEFAULT is not included in the list.

RawConfigParser.**add_section(section)**
Add a section named section to the instance. If a section by the given name already exists, DuplicateSectionError is raised. If the name DEFAULT (or any of it’s case-insensitive variants) is passed, ValueError is raised.

RawConfigParser.**has_section(section)**
Indicates whether the named section is present in the configuration. The DEFAULT section is not acknowledged.

RawConfigParser.**options(section)**
Returns a list of options available in the specified section.

RawConfigParser.**has_option(section, option)**
If the given section exists, and contains the given option, return True; otherwise return False.

*New in version 1.6.*

RawConfigParser.**read(filenames)**
Attempt to read and parse a list of filenames, returning a list of filenames which were successfully parsed. If filenames is a string or Unicode string, it is treated as a single filename. If a file named
in *filenames* cannot be opened, that file will be ignored. This is designed so that you can specify a list of potential configuration file locations (for example, the current directory, the user’s home directory, and some system-wide directory), and all existing configuration files in the list will be read. If none of the named files exist, the *ConfigParser* instance will contain an empty dataset. An application which requires initial values to be loaded from a file should load the required file or files using *readfp()* before calling *read()* for any optional files:

```
import ConfigParser, os
config = ConfigParser.ConfigParser()
config.readfp(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser '~/myapp.cfg'])
```

*Changed in version 2.4:* Returns list of successfully parsed filenames.

**RawConfigParser.*readfp*(fp[, filename])**

Read and parse configuration data from the file or file-like object in *fp* (only the *readline()* method is used). If *filename* is omitted and *fp* has a *name* attribute, that is used for *filename*; the default is `<???>`.

**RawConfigParser.*get*(section, option)**

Get an *option* value for the named *section*.

**RawConfigParser.*getint*(section, option)**

A convenience method which coerces the *option* in the specified *section* to an integer.

**RawConfigParser.*getfloat*(section, option)**

A convenience method which coerces the *option* in the specified *section* to a floating point number.
RawConfigParser.getboolean(section, option)

A convenience method which coerces the option in the specified section to a Boolean value. Note that the accepted values for the option are "1", "yes", "true", and "on", which cause this method to return True, and "0", "no", "false", and "off", which cause it to return False. These string values are checked in a case-insensitive manner. Any other value will cause it to raise ValueError.

RawConfigParser.items(section)

Return a list of (name, value) pairs for each option in the given section.

RawConfigParser.set(section, option, value)

If the given section exists, set the given option to the specified value; otherwise raise NoSectionError. While it is possible to use RawConfigParser (or ConfigParser with raw parameters set to true) for internal storage of non-string values, full functionality (including interpolation and output to files) can only be achieved using string values.

New in version 1.6.

RawConfigParser.write(fileobject)

Write a representation of the configuration to the specified file object. This representation can be parsed by a future read() call.

New in version 1.6.

RawConfigParser.remove_option(section, option)

Remove the specified option from the specified section. If the section does not exist, raise NoSectionError. If the option existed to be removed, return True; otherwise return False.
New in version 1.6.

RawConfigParser\texttt{.remove\_section}(\texttt{section})

Remove the specified \texttt{section} from the configuration. If the section in fact existed, return \texttt{True}. Otherwise return \texttt{False}.

RawConfigParser\texttt{.optionxform}(\texttt{option})

Transforms the option name \texttt{option} as found in an input file or as passed in by client code to the form that should be used in the internal structures. The default implementation returns a lower-case version of \texttt{option}; subclasses may override this or client code can set an attribute of this name on instances to affect this behavior.

You don't necessarily need to subclass a ConfigParser to use this method, you can also re-set it on an instance, to a function that takes a string argument. Setting it to \texttt{str}, for example, would make option names case sensitive:

\begin{verbatim}
cfgparser = ConfigParser()
...
cfgparser.optionxform = str
\end{verbatim}

Note that when reading configuration files, whitespace around the option names are stripped before \texttt{optionxform()} is called.
13.2.2. ConfigParser Objects

The **ConfigParser** class extends some methods of the **RawConfigParser** interface, adding some optional arguments.

ConfigParser.**get**(section, option[, raw[, vars]])

Get an *option* value for the named *section*. If *vars* is provided, it must be a dictionary. The *option* is looked up in *vars* (if provided), *section*, and in *defaults* in that order.

All the '%'-interpolations are expanded in the return values, unless the *raw* argument is true. Values for interpolation keys are looked up in the same manner as the option.

ConfigParser.**items**(section[, raw[, vars]])

Return a list of (name, value) pairs for each option in the given *section*. Optional arguments have the same meaning as for the **get()** method.

*New in version 2.3.*
13.2.3. SafeConfigParser Objects

The SafeConfigParser class implements the same extended interface as ConfigParser, with the following addition:

SafeConfigParser.set(section, option, value)

If the given section exists, set the given option to the specified value; otherwise raise NoSectionError. value must be a string (str or unicode); if not, TypeError is raised.

New in version 2.4.
13.2.4. Examples

An example of writing to a configuration file:

```python
import ConfigParser

config = ConfigParser.RawConfigParser()

# When adding sections or items, add them in the reverse order
# how you want them to be displayed in the actual file.
# In addition, please note that using RawConfigParser's and the
# mode of ConfigParser's respective set functions, you can assi
# non-string values to keys internally, but will receive an err
# when attempting to write to a file or when you get it in non-
# mode. SafeConfigParser does not allow such assignments to take
cfg.add_section('Section1')
cfg.set('Section1', 'an_int', '15')
cfg.set('Section1', 'a_bool', 'true')
cfg.set('Section1', 'a_float', '3.1415')
cfg.set('Section1', 'baz', 'fun')
cfg.set('Section1', 'bar', 'Python')
cfg.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'wb') as configfile:
    config.write(configfile)
```

An example of reading the configuration file again:

```python
import ConfigParser

config = ConfigParser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective t
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print a_float + an_int

# Notice that the next output does not interpolate '%(bar)s' or
# This is because we are using a RawConfigParser().
```
if config.getboolean('Section1', 'a_bool'):
    print config.get('Section1', 'foo')

To get interpolation, you will need to use a `ConfigParser` or `SafeConfigParser`:

```python
import ConfigParser

config = ConfigParser()
config.read('example.cfg')

# Set the third, optional argument of get to 1 if you wish to use interpolation.
print config.get('Section1', 'foo', 0) # -> "Python is fun!"
print config.get('Section1', 'foo', 1) # -> "%(bar)s is %(baz)s!"

# The optional fourth argument is a dict with members that will take precedence in interpolation.
print config.get('Section1', 'foo', 0, {'bar': 'Documentation', 'baz': 'evil'})
```

Defaults are available in all three types of ConfigParsers. They are used in interpolation if an option used is not defined elsewhere.

```python
import ConfigParser

# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each.
config = ConfigParser.SafeConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print config.get('Section1', 'foo') # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print config.get('Section1', 'foo') # -> "Life is hard!"
```

The function `opt_move` below can be used to move options between sections:

```python
def opt_move(config, section1, section2, option):
    try:
        config.remove_option(section1, option)
```
Some configuration files are known to include settings without values, but which otherwise conform to the syntax supported by ConfigParser. The allow_no_value parameter to the constructor can be used to indicate that such values should be accepted:

```python
>>> import ConfigParser
>>> import io

>>> sample_config = ""
... [mysqld]
... user = mysql
... pid-file = /var/run/mysqld/mysqld.pid
... skip-external-locking
... old_passwords = 1
... skip-bdb
... skip-innodb
... ""
>>> config = ConfigParser.RawConfigParser(allow_no_value=True)
>>> config.readfp(io.BytesIO(sample_config))

>>> # Settings with values are treated as before:
>>> config.get("mysqld", "user")
'mysql'

>>> # Settings without values provide None:
>>> config.get("mysqld", "skip-bdb")

>>> # Settings which aren't specified still raise an error:
>>> config.get("mysqld", "does-not-exist")
Traceback (most recent call last):
  ...  
ConfigParser.NoOptionError: No option 'does-not-exist' in section: 'mysqld'
```
13.3. robotparser — Parser for robots.txt

Note: The robotparser module has been renamed urllib.robotparser in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

This module provides a single class, RobotFileParser, which answers questions about whether or not a particular user agent can fetch a URL on the Web site that published the robots.txt file. For more details on the structure of robots.txt files, see http://www.robotstxt.org/orig.html.

class robotparser.RobotFileParser(url='')

This class provides methods to read, parse and answer questions about the robots.txt file at url.

set_url(url)

Sets the URL referring to a robots.txt file.

read()

Reads the robots.txt URL and feeds it to the parser.

parse(lines)

Parses the lines argument.

can_fetch(useragent, url)

Returns True if the useragent is allowed to fetch the url according to the rules contained in the parsed robots.txt file.

mtime()

Returns the time the robots.txt file was last fetched. This is
useful for long-running web spiders that need to check for new `robots.txt` files periodically.

`modified()`
Sets the time the `robots.txt` file was last fetched to the current time.

The following example demonstrates basic use of the RobotFileParser class.

```python
>>> import robotparser
>>> rp = robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?c")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```
13.4. **netrc** — netrc file processing

*New in version 1.5.2.*

**Source code:** Lib/netrc.py

The `netrc` class parses and encapsulates the netrc file format used by the Unix `ftp` program and other FTP clients.

```python
class netrc.netrc[file]
```

A `netrc` instance or subclass instance encapsulates data from a netrc file. The initialization argument, if present, specifies the file to parse. If no argument is given, the file `.netrc` in the user’s home directory will be read. Parse errors will raise `NetrcParseError` with diagnostic information including the file name, line number, and terminating token. If no argument is specified on a POSIX system, the presence of passwords in the `.netrc` file will raise a `NetrcParseError` if the file ownership or permissions are insecure (owned by a user other than the user running the process, or accessible for read or write by any other user). This implements security behavior equivalent to that of `ftp` and other programs that use `.netrc`.

*Changed in version 2.7.6:* Added the POSIX permissions check.

**Exception** `netrc.NetrcParseError`

Exception raised by the `netrc` class when syntactical errors are encountered in source text. Instances of this exception provide three interesting attributes: `msg` is a textual explanation of the error, `filename` is the name of the source file, and `lineno` gives the line number on which the error was found.
13.4.1. netrc Objects

A `netrc` instance has the following methods:

`netrc.authenticators(host)`

Return a 3-tuple `(login, account, password)` of authenticators for `host`. If the netrc file did not contain an entry for the given host, return the tuple associated with the 'default' entry. If neither matching host nor default entry is available, return `None`.

`netrc.__repr__()`

Dump the class data as a string in the format of a netrc file. (This discards comments and may reorder the entries.)

Instances of `netrc` have public instance variables:

`netrc.hosts`

Dictionary mapping host names to `(login, account, password)` tuples. The 'default' entry, if any, is represented as a pseudo-host by that name.

`netrc.macros`

Dictionary mapping macro names to string lists.

**Note:** Passwords are limited to a subset of the ASCII character set. Versions of this module prior to 2.3 were extremely limited. Starting with 2.3, all ASCII punctuation is allowed in passwords. However, note that whitespace and non-printable characters are not allowed in passwords. This is a limitation of the way the .netrc file is parsed and may be removed in the future.
The *xdrlib* module supports the External Data Representation Standard as described in RFC 1014, written by Sun Microsystems, Inc. June 1987. It supports most of the data types described in the RFC.

The *xdrlib* module defines two classes, one for packing variables into XDR representation, and another for unpacking from XDR representation. There are also two exception classes.

```python
class xdrlib.Packer
    Packer is the class for packing data into XDR representation. The Packer class is instantiated with no arguments.

class xdrlib.Unpacker(data)
    Unpacker is the complementary class which unpacks XDR data values from a string buffer. The input buffer is given as `data`.
```

See also:

**RFC 1014 - XDR: External Data Representation Standard**

This RFC defined the encoding of data which was XDR at the time this module was originally written. It has apparently been obsoleted by RFC 1832.

**RFC 1832 - XDR: External Data Representation Standard**

Newer RFC that provides a revised definition of XDR.
13.5.1. Packer Objects

**Packer** instances have the following methods:

**Packer.get_buffer()**

Returns the current pack buffer as a string.

**Packer.reset()**

Resets the pack buffer to the empty string.

In general, you can pack any of the most common XDR data types by calling the appropriate `pack_type()` method. Each method takes a single argument, the value to pack. The following simple data type packing methods are supported: `pack_uint()`, `pack_int()`, `pack_enum()`, `pack_bool()`, `pack_uhyper()`, and `pack_hyper()`.

**Packer.pack_float(value)**

Packs the single-precision floating point number `value`.

**Packer.pack_double(value)**

Packs the double-precision floating point number `value`.

The following methods support packing strings, bytes, and opaque data:

**Packer.pack_fstring(n, s)**

Packs a fixed length string, `s`. `n` is the length of the string but it is *not* packed into the data buffer. The string is padded with null bytes if necessary to guaranteed 4 byte alignment.

**Packer.pack_fopaque(n, data)**

Packs a fixed length opaque data stream, similarly to `pack_fstring()`.
**Packer.** `.pack_string(s)`

Packs a variable length string, `s`. The length of the string is first packed as an unsigned integer, then the string data is packed with `.pack_fstring()`.

**Packer.** `.pack_opaque(data)`

Packs a variable length opaque data string, similarly to `.pack_string()`.

**Packer.** `.pack_bytes(bytes)`

Packs a variable length byte stream, similarly to `.pack_string()`.

The following methods support packing arrays and lists:

**Packer.** `.pack_list(list, pack_item)`

Packs a list of homogeneous items. This method is useful for lists with an indeterminate size; i.e. the size is not available until the entire list has been walked. For each item in the list, an unsigned integer 1 is packed first, followed by the data value from the list. `.pack_item` is the function that is called to pack the individual item. At the end of the list, an unsigned integer 0 is packed.

For example, to pack a list of integers, the code might appear like this:

```python
import xdrlib
p = xdrlib.Packer()
p.pack_list([1, 2, 3], p.pack_int)
```

**Packer.** `.pack_farray(n, array, pack_item)`

Packs a fixed length list (array) of homogeneous items. `n` is the length of the list; it is not packed into the buffer, but a `ValueError` exception is raised if `len(array)` is not equal to `n`. As above, `.pack_item` is the function used to pack each element.
Packer. **pack_array**(list, pack_item)

Packs a variable length *list* of homogeneous items. First, the length of the list is packed as an unsigned integer, then each element is packed as in *pack_farray()* above.
13.5.2. Unpacker Objects

The `Unpacker` class offers the following methods:

- **Unpacker.reset(data)**
  Resets the string buffer with the given `data`.

- **Unpacker.get_position()**
  Returns the current unpack position in the data buffer.

- **Unpacker.set_position(position)**
  Sets the data buffer unpack position to `position`. You should be careful about using `get_position()` and `set_position()`.

- **Unpacker.get_buffer()**
  Returns the current unpack data buffer as a string.

- **Unpacker.done()**
  Indicates unpack completion. Raises an `Error` exception if all of the data has not been unpacked.

In addition, every data type that can be packed with a `Packer`, can be unpacked with an `Unpacker`. Unpacking methods are of the form `unpack_type()`, and take no arguments. They return the unpacked object.

- **Unpacker.unpack_float()**
  Unpacks a single-precision floating point number.

- **Unpacker.unpack_double()**
  Unpacks a double-precision floating point number, similarly to `unpack_float()`.

In addition, the following methods unpack strings, bytes, and opaque
data:

**Unpacker.** **unpack_fstring**(*n*)
Unpacks and returns a fixed length string. *n* is the number of characters expected. Padding with null bytes to guaranteed 4 byte alignment is assumed.

**Unpacker.** **unpack_fopaque**(*n*)
Unpacks and returns a fixed length opaque data stream, similarly to **unpack_fstring**().

**Unpacker.** **unpack_string**()
Unpacks and returns a variable length string. The length of the string is first unpacked as an unsigned integer, then the string data is unpacked with **unpack_fstring**().

**Unpacker.** **unpack_opaque**()
Unpacks and returns a variable length opaque data string, similarly to **unpack_string**().

**Unpacker.** **unpack_bytes**()
Unpacks and returns a variable length byte stream, similarly to **unpack_string**().

The following methods support unpacking arrays and lists:

**Unpacker.** **unpack_list**(unpack item)
Unpacks and returns a list of homogeneous items. The list is unpacked one element at a time by first unpacking an unsigned integer flag. If the flag is 1, then the item is unpacked and appended to the list. A flag of 0 indicates the end of the list. **unpack_item** is the function that is called to unpack the items.

**Unpacker.** **unpack_farray**(n, unpack item)
Unpacks and returns (as a list) a fixed length array of
homogeneous items. \( n \) is number of list elements to expect in the buffer. As above, \texttt{unpack\_item} is the function used to unpack each element.

\textbf{Unpacker. \texttt{unpack\_array}(unpack\_item)}

Unpacks and returns a variable length list of homogeneous items. First, the length of the list is unpacked as an unsigned integer, then each element is unpacked as in \texttt{unpack\_farray()} above.
13.5.3. Exceptions

Exceptions in this module are coded as class instances:

```python
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError as instance:
    print 'packing the double failed:', instance.msg
```

**exception** `xdrlib.Error`

The base exception class. `Error` has a single public attribute `msg` containing the description of the error.

**exception** `xdrlib.ConversionError`

Class derived from `Error`. Contains no additional instance variables.

Here is an example of how you would catch one of these exceptions:
This module provides an interface for reading and writing the “property list” XML files used mainly by Mac OS X.

The property list (.plist) file format is a simple XML pickle supporting basic object types, like dictionaries, lists, numbers and strings. Usually the top level object is a dictionary.

Values can be strings, integers, floats, booleans, tuples, lists, dictionaries (but only with string keys), Data or datetime.datetime objects. String values (including dictionary keys) may be unicode strings – they will be written out as UTF-8.

The <data> plist type is supported through the Data class. This is a thin wrapper around a Python string. Use Data if your strings contain control characters.

**See also:**

PList manual page

Apple's documentation of the file format.

This module defines the following functions:

`plistlib.readPlist(pathOrFile)`
Read a plist file. `pathOrFile` may either be a file name or a (readable) file object. Return the unpacked root object (which usually is a dictionary).

The XML data is parsed using the Expat parser from `xml.parsers.expat` – see its documentation for possible exceptions on ill-formed XML. Unknown elements will simply be ignored by the plist parser.

`plistlib.writePlist(rootObject, pathOrFile)`
Write `rootObject` to a plist file. `pathOrFile` may either be a file name or a (writable) file object.

A `TypeError` will be raised if the object is of an unsupported type or a container that contains objects of unsupported types.

`plistlib.readPlistFromString(data)`
Read a plist from a string. Return the root object.

`plistlib.writePlistToString(rootObject)`
Return `rootObject` as a plist-formatted string.

`plistlib.readPlistFromResource(path, restype='plst', resid=0)`
Read a plist from the resource with type `restype` from the resource fork of `path`. Availability: Mac OS X.

**Note:** In Python 3.x, this function has been removed.

`plistlib.writePlistToResource(rootObject, path, restype='plst', resid=0)`
Write `rootObject` as a resource with type `restype` to the resource fork of `path`. Availability: Mac OS X.

**Note:** In Python 3.x, this function has been removed.
The following class is available:

```python
class plistlib.Data(data)
    Return a “data” wrapper object around the string data. This is used in functions converting from/to plists to represent the <data> type available in plists.
```

It has one attribute, `data`, that can be used to retrieve the Python string stored in it.
### 13.6.1. Examples

#### Generating a plist:

```python
pl = dict(
aString="Doodah",
aList=["A", "B", 12, 32.1, [1, 2, 3]],
aFloat = 0.1,
anInt = 728,
aDict=dict(
    anotherString="<hello & hi there!>",
aUnicodeValue=u'M\xe4ssig, Ma\xdf',
aTrueValue=True,
aFalseValue=False,
),
someData = Data("<binary gunk>"),
someMoreData = Data("<lots of binary gunk>" * 10),
aDate = datetime.datetime.fromtimestamp(time.mktime(time.gmtime()))
)
# unicode keys are possible, but a little awkward to use:
pl[u'\xc5benraa'] = "That was a unicode key."
writePlist(pl, fileName)
```

#### Parsing a plist:

```python
pl = readPlist(pathOrFile)
print pl["aKey"]
```
14. Cryptographic Services

The modules described in this chapter implement various algorithms of a cryptographic nature. They are available at the discretion of the installation. Here’s an overview:

- **14.1. hashlib** — Secure hashes and message digests
  - 14.1.1. Key derivation
- **14.2. hmac** — Keyed-Hashing for Message Authentication
- **14.3. md5** — MD5 message digest algorithm
- **14.4. sha** — SHA-1 message digest algorithm
14.1. `hashlib` — Secure hashes and message digests

`New in version 2.5.`

**Source code:** Lib/hashlib.py

This module implements a common interface to many different secure hash and message digest algorithms. Included are the FIPS secure hash algorithms SHA1, SHA224, SHA256, SHA384, and SHA512 (defined in FIPS 180-2) as well as RSA's MD5 algorithm (defined in Internet [RFC 1321](https://www.rfc-editor.org/rfc/rfc1321)). The terms secure hash and message digest are interchangeable. Older algorithms were called message digests. The modern term is secure hash.

**Note:** If you want the adler32 or crc32 hash functions, they are available in the `zlib` module.

**Warning:** Some algorithms have known hash collision weaknesses, refer to the “See also” section at the end.

There is one constructor method named for each type of `hash`. All return a hash object with the same simple interface. For example: use `sha1()` to create a SHA1 hash object. You can now feed this object with arbitrary strings using the `update()` method. At any point you can ask it for the `digest` of the concatenation of the strings fed to it so far using the `digest()` or `hexdigest()` methods.

Constructors for hash algorithms that are always present in this module are `md5()`, `sha1()`, `sha224!()`, `sha256()`, `sha384()`, and `sha512()`. Additional algorithms may also be available depending
upon the OpenSSL library that Python uses on your platform.

For example, to obtain the digest of the string 'Nobody inspects the spammish repetition':

```python
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.update("the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
>>> m.digest_size
16
>>> m.block_size
64
```

More condensed:

```python
>>> hashlib.sha224("Nobody inspects the spammish repetition").hexdigest()
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

A generic `new()` constructor that takes the string name of the desired algorithm as its first parameter also exists to allow access to the above listed hashes as well as any other algorithms that your OpenSSL library may offer. The named constructors are much faster than `new()` and should be preferred.

Using `new()` with an algorithm provided by OpenSSL:

```python
>>> h = hashlib.new('ripemd160')
>>> h.update("Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

This module provides the following constant attribute:

`hashlib.algorithms`
A tuple providing the names of the hash algorithms guaranteed to be supported by this module.

*New in version 2.7.*

`hashlib.algorithms_guaranteed`

A set containing the names of the hash algorithms guaranteed to be supported by this module on all platforms.

*New in version 2.7.9.*

`hashlib.algorithms_available`

A set containing the names of the hash algorithms that are available in the running Python interpreter. These names will be recognized when passed to `new()`. `algorithms_guaranteed` will always be a subset. The same algorithm may appear multiple times in this set under different names (thanks to OpenSSL).

*New in version 2.7.9.*

The following values are provided as constant attributes of the hash objects returned by the constructors:

`hash.digest_size`

The size of the resulting hash in bytes.

`hash.block_size`

The internal block size of the hash algorithm in bytes.

A hash object has the following methods:

`hash.update(arg)`

Update the hash object with the string `arg`. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a+b)`. 
*Changed in version 2.7:* The Python GIL is released to allow other threads to run while hash updates on data larger than 2048 bytes is taking place when using hash algorithms supplied by OpenSSL.

**hash. `digest()`**
Return the digest of the strings passed to the `update()` method so far. This is a string of `digest_size` bytes which may contain non-ASCII characters, including null bytes.

**hash. `hexdigest()`**
Like `digest()` except the digest is returned as a string of double length, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

**hash. `copy()`**
Return a copy ("clone") of the hash object. This can be used to efficiently compute the digests of strings that share a common initial substring.
14.1.1. Key derivation

Key derivation and key stretching algorithms are designed for secure password hashing. Naive algorithms such as `sha1(password)` are not resistant against brute-force attacks. A good password hashing function must be tunable, slow, and include a `salt`.

```python
import hashlib, binascii

dk = hashlib.pbkdf2_hmac('sha256', b'password', b'salt', 100000)
b'0394a2ede32c9a13eb82e9b24631604c31df978b4e2f0fbd2c549944f
```

**Note:** A fast implementation of `pbkdf2_hmac` is available with OpenSSL. The Python implementation uses an inline version of
hmac. It is about three times slower and doesn’t release the GIL.

See also:

**Module hmac**
A module to generate message authentication codes using hashes.

**Module base64**
Another way to encode binary hashes for non-binary environments.

The FIPS 180-2 publication on Secure Hash Algorithms.

https://en.wikipedia.org/wiki/Cryptographic_hash_function#Cryptographic_hash_algorithms
Wikipedia article with information on which algorithms have known issues and what that means regarding their use.
14.2. hmac — Keyed-Hashing for Message Authentication

New in version 2.2.

Source code: Lib/hmac.py

This module implements the HMAC algorithm as described by RFC 2104.

```
import hmac

hmac.new(key[, msg[, digestmod]])
```

Return a new hmac object. If `msg` is present, the method call `update(msg)` is made. `digestmod` is the digest constructor or module for the HMAC object to use. It defaults to the `hashlib.md5` constructor.

An HMAC object has the following methods:

```
HMAC.update(msg)
```

Update the hmac object with the string `msg`. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a + b)`.

```
HMAC.digest()
```

Return the digest of the strings passed to the `update()` method so far. This string will be the same length as the `digest_size` of the digest given to the constructor. It may contain non-ASCII characters, including NUL bytes.

**Warning:** When comparing the output of `digest()` to an externally-supplied digest during a verification routine, it is
recommended to use the `compare_digest()` function instead of the `==` operator to reduce the vulnerability to timing attacks.

**HMAC.hexdigest()**

Like `digest()` except the digest is returned as a string twice the length containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

**Warning:** When comparing the output of `hexdigest()` to an externally-supplied digest during a verification routine, it is recommended to use the `compare_digest()` function instead of the `==` operator to reduce the vulnerability to timing attacks.

**HMAC.copy()**

Return a copy (“clone”) of the hmac object. This can be used to efficiently compute the digests of strings that share a common initial substring.

This module also provides the following helper function:

**hmac.compare_digest(a, b)**

Return `a == b`. This function uses an approach designed to prevent timing analysis by avoiding content-based short circuiting behaviour, making it appropriate for cryptography. `a` and `b` must both be of the same type: either `unicode` or a `bytes-like object`.

**Note:** If `a` and `b` are of different lengths, or if an error occurs, a timing attack could theoretically reveal information about the types and lengths of `a` and `b—but not their values.

*New in version 2.7.7.*
See also:

**Module `hashlib`**

The Python module providing secure hash functions.
14.3. **md5** — MD5 message digest algorithm

*Deprecated since version 2.5: Use the `hashlib` module instead.*

This module implements the interface to RSA's MD5 message digest algorithm (see also Internet RFC 1321). Its use is quite straightforward: use `new()` to create an md5 object. You can now feed this object with arbitrary strings using the `update()` method, and at any point you can ask it for the *digest* (a strong kind of 128-bit checksum, a.k.a. “fingerprint”) of the concatenation of the strings fed to it so far using the `digest()` method.

For example, to obtain the digest of the string "Nobody inspects the spammish repetition":

```python
>>> import md5
>>> m = md5.new()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
```

More condensed:

```python
>>> md5.new("Nobody inspects the spammish repetition").digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
```

The following values are provided as constants in the module and as attributes of the md5 objects returned by `new()`:

```python
md5.digest_size
```
The size of the resulting digest in bytes. This is always 16.

The md5 module provides the following functions:

\texttt{md5.new([arg])}

Return a new md5 object. If \texttt{arg} is present, the method call \texttt{update(arg)} is made.

\texttt{md5([arg])}

For backward compatibility reasons, this is an alternative name for the \texttt{new()} function.

An md5 object has the following methods:

\texttt{md5.update(arg)}

Update the md5 object with the string \texttt{arg}. Repeated calls are equivalent to a single call with the concatenation of all the arguments: \texttt{m.update(a); m.update(b)} is equivalent to \texttt{m.update(a+b)}.

\texttt{md5.digest()}

Return the digest of the strings passed to the \texttt{update()} method so far. This is a 16-byte string which may contain non-ASCII characters, including null bytes.

\texttt{md5.hexdigest()}

Like \texttt{digest()} except the digest is returned as a string of length 32, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

\texttt{md5.copy()}

Return a copy (“clone”) of the md5 object. This can be used to efficiently compute the digests of strings that share a common
initial substring.

See also:

Module sha

Similar module implementing the Secure Hash Algorithm (SHA). The SHA algorithm is considered a more secure hash.
14.4. sha — SHA-1 message digest algorithm

Deprecated since version 2.5: Use the hashlib module instead.

This module implements the interface to NIST’s secure hash algorithm, known as SHA-1. SHA-1 is an improved version of the original SHA hash algorithm. It is used in the same way as the md5 module: use new() to create an sha object, then feed this object with arbitrary strings using the update() method, and at any point you can ask it for the digest of the concatenation of the strings fed to it so far. SHA-1 digests are 160 bits instead of MD5’s 128 bits.

```
sha.new([string])
```

Return a new sha object. If string is present, the method call update(string) is made.

The following values are provided as constants in the module and as attributes of the sha objects returned by new():

```
sha.blocksize
```

Size of the blocks fed into the hash function; this is always 1. This size is used to allow an arbitrary string to be hashed.

```
sha.digest_size
```

The size of the resulting digest in bytes. This is always 20.

An sha object has the same methods as md5 objects:

```
sha.update(arg)
```

Update the sha object with the string arg. Repeated calls are equivalent to a single call with the concatenation of all the
arguments: \texttt{m.update(a); m.update(b)} is equivalent to \texttt{m.update(a+b)}.

\texttt{sha.digest()}

Return the digest of the strings passed to the \texttt{update()} method so far. This is a 20-byte string which may contain non-ASCII characters, including null bytes.

\texttt{sha.hexdigest()}

Like \texttt{digest()} except the digest is returned as a string of length 40, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

\texttt{sha.copy()}

Return a copy ("clone") of the sha object. This can be used to efficiently compute the digests of strings that share a common initial substring.

\textbf{See also:}

\textbf{Secure Hash Standard}


\textbf{Cryptographic Toolkit (Secure Hashing)}

Links from NIST to various information on secure hashing.
15. Generic Operating System Services

The modules described in this chapter provide interfaces to operating system features that are available on (almost) all operating systems, such as files and a clock. The interfaces are generally modeled after the Unix or C interfaces, but they are available on most other systems as well. Here's an overview:

- **15.1. os** — Miscellaneous operating system interfaces
  - 15.1.1. Process Parameters
  - 15.1.2. File Object Creation
  - 15.1.3. File Descriptor Operations
    - 15.1.3.1. `open()` flag constants
  - 15.1.4. Files and Directories
  - 15.1.5. Process Management
  - 15.1.6. Miscellaneous System Information
  - 15.1.7. Miscellaneous Functions
- **15.2. io** — Core tools for working with streams
  - 15.2.1. Module Interface
  - 15.2.2. I/O Base Classes
  - 15.2.3. Raw File I/O
  - 15.2.4. Buffered Streams
  - 15.2.5. Text I/O
  - 15.2.6. Advanced topics
    - 15.2.6.1. Performance
      - 15.2.6.1.1. Binary I/O
      - 15.2.6.1.2. Text I/O
    - 15.2.6.2. Multi-threading
    - 15.2.6.3. Reentrancy
- **15.3. time** — Time access and conversions
- **15.4. argparse** — Parser for command-line options, arguments
and sub-commands

- **15.4.1. Example**
  - 15.4.1.1. Creating a parser
  - 15.4.1.2. Adding arguments
  - 15.4.1.3. Parsing arguments

- **15.4.2. ArgumentParser objects**
  - 15.4.2.1. prog
  - 15.4.2.2. usage
  - 15.4.2.3. description
  - 15.4.2.4. epilog
  - 15.4.2.5. parents
  - 15.4.2.6. formatter_class
  - 15.4.2.7. prefix_chars
  - 15.4.2.8. fromfile_prefix_chars
  - 15.4.2.9. argument_default
  - 15.4.2.10. conflict_handler
  - 15.4.2.11. add_help

- **15.4.3. The add_argument() method**
  - 15.4.3.1. name or flags
  - 15.4.3.2. action
  - 15.4.3.3. nargs
  - 15.4.3.4. const
  - 15.4.3.5. default
  - 15.4.3.6. type
  - 15.4.3.7. choices
  - 15.4.3.8. required
  - 15.4.3.9. help
  - 15.4.3.10. metavar
  - 15.4.3.11. dest
  - 15.4.3.12. Action classes

- **15.4.4. The parse_args() method**
  - 15.4.4.1. Option value syntax
  - 15.4.4.2. Invalid arguments
  - 15.4.4.3. Arguments containing -
- 15.4.4.4. Argument abbreviations (prefix matching)
- 15.4.4.5. Beyond `sys.argv`
- 15.4.4.6. The Namespace object
- **15.4.5. Other utilities**
  - 15.4.5.1. Sub-commands
  - 15.4.5.2. FileType objects
  - 15.4.5.3. Argument groups
  - 15.4.5.4. Mutual exclusion
  - 15.4.5.5. Parser defaults
  - 15.4.5.6. Printing help
  - 15.4.5.7. Partial parsing
  - 15.4.5.8. Customizing file parsing
  - 15.4.5.9. Exiting methods
- **15.4.6. Upgrading optparse code**
- **15.5. `optparse` — Parser for command line options**
  - **15.5.1. Background**
    - 15.5.1.1. Terminology
    - 15.5.1.2. What are options for?
    - 15.5.1.3. What are positional arguments for?
  - **15.5.2. Tutorial**
    - 15.5.2.1. Understanding option actions
    - 15.5.2.2. The store action
    - 15.5.2.3. Handling boolean (flag) options
    - 15.5.2.4. Other actions
    - 15.5.2.5. Default values
    - 15.5.2.6. Generating help
      - 15.5.2.6.1. Grouping Options
    - 15.5.2.7. Printing a version string
    - 15.5.2.8. How `optparse` handles errors
    - 15.5.2.9. Putting it all together
  - **15.5.3. Reference Guide**
    - 15.5.3.1. Creating the parser
    - 15.5.3.2. Populating the parser
    - 15.5.3.3. Defining options
- 15.5.3.4. Option attributes
- 15.5.3.5. Standard option actions
- 15.5.3.6. Standard option types
- 15.5.3.7. Parsing arguments
- 15.5.3.8. Querying and manipulating your option parser
- 15.5.3.9. Conflicts between options
- 15.5.3.10. Cleanup
- 15.5.3.11. Other methods

  - 15.5.4. Option Callbacks
    - 15.5.4.1. Defining a callback option
    - 15.5.4.2. How callbacks are called
    - 15.5.4.3. Raising errors in a callback
    - 15.5.4.4. Callback example 1: trivial callback
    - 15.5.4.5. Callback example 2: check option order
    - 15.5.4.6. Callback example 3: check option order (generalized)
    - 15.5.4.7. Callback example 4: check arbitrary condition
    - 15.5.4.8. Callback example 5: fixed arguments
    - 15.5.4.9. Callback example 6: variable arguments

  - 15.5.5. Extending optparse
    - 15.5.5.1. Adding new types
    - 15.5.5.2. Adding new actions

- 15.6. getopt — C-style parser for command line options

- 15.7. logging — Logging facility for Python
  - 15.7.1. Logger Objects
  - 15.7.2. Logging Levels
  - 15.7.3. Handler Objects
  - 15.7.4. Formatter Objects
  - 15.7.5. Filter Objects
  - 15.7.6. LogRecord Objects
  - 15.7.7. LogRecord attributes
  - 15.7.8. LoggerAdapter Objects
  - 15.7.9. Thread Safety
  - 15.7.10. Module-Level Functions
- 15.7.11. Integration with the warnings module
- **15.8. `logging.config` — Logging configuration**
  - 15.8.1. Configuration functions
  - 15.8.2. Configuration dictionary schema
    - 15.8.2.1. Dictionary Schema Details
    - 15.8.2.2. Incremental Configuration
    - 15.8.2.3. Object connections
    - 15.8.2.4. User-defined objects
    - 15.8.2.5. Access to external objects
    - 15.8.2.6. Access to internal objects
    - 15.8.2.7. Import resolution and custom importers
  - 15.8.3. Configuration file format
- **15.9. `logging.handlers` — Logging handlers**
  - 15.9.1. StreamHandler
  - 15.9.2. FileHandler
  - 15.9.3. NullHandler
  - 15.9.4. WatchedFileHandler
  - 15.9.5. RotatingFileHandler
  - 15.9.6. TimedRotatingFileHandler
  - 15.9.7. SocketHandler
  - 15.9.8. DatagramHandler
  - 15.9.9. SysLogHandler
  - 15.9.10. NTEventLogHandler
  - 15.9.11. SMTPHandler
  - 15.9.12. MemoryHandler
  - 15.9.13. HTTPHandler
- 15.10. `getpass` — Portable password input
- 15.11. `curses` — Terminal handling for character-cell displays
  - 15.11.1. Functions
  - 15.11.2. Window Objects
  - 15.11.3. Constants
- **15.12. `curses.textpad` — Text input widget for curses programs**
  - 15.12.1. Textbox objects
• 15.13. `curses.ascii` — Utilities for ASCII characters
• 15.14. `curses.panel` — A panel stack extension for curses
  - 15.14.1. Functions
  - 15.14.2. Panel Objects
• 15.15. `platform` — Access to underlying platform’s identifying data
  - 15.15.1. Cross Platform
  - 15.15.2. Java Platform
  - 15.15.3. Windows Platform
    - 15.15.3.1. Win95/98 specific
  - 15.15.4. Mac OS Platform
  - 15.15.5. Unix Platforms
• 15.16. `errno` — Standard errno system symbols
• 15.17. `ctypes` — A foreign function library for Python
  - 15.17.1. ctypes tutorial
    - 15.17.1.1. Loading dynamic link libraries
    - 15.17.1.2. Accessing functions from loaded dlls
    - 15.17.1.3. Calling functions
    - 15.17.1.4. Fundamental data types
    - 15.17.1.5. Calling functions, continued
    - 15.17.1.6. Calling functions with your own custom data types
    - 15.17.1.7. Specifying the required argument types (function prototypes)
    - 15.17.1.8. Return types
    - 15.17.1.9. Passing pointers (or: passing parameters by reference)
    - 15.17.1.10. Structures and unions
    - 15.17.1.11. Structure/union alignment and byte order
    - 15.17.1.12. Bit fields in structures and unions
    - 15.17.1.13. Arrays
    - 15.17.1.15. Type conversions
    - 15.17.1.16. Incomplete Types
- 15.17.1.17. Callback functions
- 15.17.1.18. Accessing values exported from dlls
- 15.17.1.19. Surprises
- 15.17.1.20. Variable-sized data types
- 15.17.2. ctypes reference
  - 15.17.2.1. Finding shared libraries
  - 15.17.2.2. Loading shared libraries
  - 15.17.2.3. Foreign functions
  - 15.17.2.4. Function prototypes
  - 15.17.2.5. Utility functions
  - 15.17.2.6. Data types
  - 15.17.2.7. Fundamental data types
  - 15.17.2.8. Structured data types
  - 15.17.2.9. Arrays and pointers
15.1. os — Miscellaneous operating system interfaces

This module provides a portable way of using operating system dependent functionality. If you just want to read or write a file see `open()`, if you want to manipulate paths, see the `os.path` module, and if you want to read all the lines in all the files on the command line see the `fileinput` module. For creating temporary files and directories see the `tempfile` module, and for high-level file and directory handling see the `shutil` module.

Notes on the availability of these functions:

- The design of all built-in operating system dependent modules of Python is such that as long as the same functionality is available, it uses the same interface; for example, the function `os.stat(path)` returns stat information about `path` in the same format (which happens to have originated with the POSIX interface).
- Extensions peculiar to a particular operating system are also available through the `os` module, but using them is of course a threat to portability.
- An “Availability: Unix” note means that this function is commonly found on Unix systems. It does not make any claims about its existence on a specific operating system.
- If not separately noted, all functions that claim “Availability: Unix” are supported on Mac OS X, which builds on a Unix core.

Note: All functions in this module raise `OSError` in the case of invalid or inaccessible file names and paths, or other arguments that have the correct type, but are not accepted by the operating system.
**exception** `os.error`

An alias for the built-in `OSError` exception.

**os.name**

The name of the operating system dependent module imported. The following names have currently been registered: `

See also: `sys.platform` has a finer granularity. `os.uname()` gives system-dependent version information. The `platform` module provides detailed checks for the system's identity.
15.1.1. Process Parameters

These functions and data items provide information and operate on the current process and user.

**os.environ**

A mapping object representing the string environment. For example, `environ['HOME']` is the pathname of your home directory (on some platforms), and is equivalent to `getenv("HOME")` in C.

This mapping is captured the first time the `os` module is imported, typically during Python startup as part of processing `site.py`. Changes to the environment made after this time are not reflected in `os.environ`, except for changes made by modifying `os.environ` directly.

If the platform supports the `putenv()` function, this mapping may be used to modify the environment as well as query the environment. `putenv()` will be called automatically when the mapping is modified.

**Note:** Calling `putenv()` directly does not change `os.environ`, so it's better to modify `os.environ`.

**Note:** On some platforms, including FreeBSD and Mac OS X, setting `environ` may cause memory leaks. Refer to the system documentation for `putenv()`.

If `putenv()` is not provided, a modified copy of this mapping may be passed to the appropriate process-creation functions to cause child processes to use a modified environment.
If the platform supports the `unsetenv()` function, you can delete items in this mapping to unset environment variables. `unsetenv()` will be called automatically when an item is deleted from `os.environ`, and when one of the `pop()` or `clear()` methods is called.

*Changed in version 2.6:* Also unset environment variables when calling `os.environ.clear()` and `os.environ.pop()`.

- `os.chdir(path)`
- `os.fchdir(fd)`
- `os.getcwd()`

These functions are described in *Files and Directories.*

- `os.ctermid()`
  Return the filename corresponding to the controlling terminal of the process.

  Availability: Unix.

- `os.getegid()`
  Return the effective group id of the current process. This corresponds to the “set id” bit on the file being executed in the current process.

  Availability: Unix.

- `os.geteuid()`
  Return the current process’s effective user id.

  Availability: Unix.

- `os.getgid()`
  Return the real group id of the current process.

  Availability: Unix.
os. `getgroups()`

Return list of supplemental group ids associated with the current process.

Availability: Unix.

**Note:** On Mac OS X, `getgroups()` behavior differs somewhat from other Unix platforms. If the Python interpreter was built with a deployment target of 10.5 or earlier, `getgroups()` returns the list of effective group ids associated with the current user process; this list is limited to a system-defined number of entries, typically 16, and may be modified by calls to `setgroups()` if suitably privileged. If built with a deployment target greater than 10.5, `getgroups()` returns the current group access list for the user associated with the effective user id of the process; the group access list may change over the lifetime of the process, it is not affected by calls to `setgroups()`, and its length is not limited to 16. The deployment target value, `MACOSX_DEPLOYMENT_TARGET`, can be obtained with `sysconfig.get_config_var()`.

os. `initgroups(username, gid)`

Call the system `initgroups()` to initialize the group access list with all of the groups of which the specified username is a member, plus the specified group id.

Availability: Unix.

*New in version 2.7.*

os. `getlogin()`

Return the name of the user logged in on the controlling terminal of the process. For most purposes, it is more useful to use the environment variable `LOGNAME` to find out who the user is, or
pwd.getpwuid(os.getuid())[0] to get the login name of the process's real user id.

Availability: Unix.

`os.getpgid(pid)`
Return the process group id of the process with process id `pid`. If `pid` is 0, the process group id of the current process is returned.

Availability: Unix.

*New in version 2.3.*

`os.getpgrp()`
Return the id of the current process group.

Availability: Unix.

`os.getpid()`
Return the current process id.

Availability: Unix, Windows.

`os.getppid()`
Return the parent's process id.

Availability: Unix.

`os.getresuid()`
Return a tuple (ruid, euid, suid) denoting the current process's real, effective, and saved user ids.

Availability: Unix.

*New in version 2.7.*

`os.getresgid()`
Return a tuple (rgid, egid, sgid) denoting the current process's real, effective, and saved group ids.

Availability: Unix.

*New in version 2.7.*

```python
os.getuid()
```
Return the current process's real user id.

Availability: Unix.

```python
os.getenv(varname[, value])
```
Return the value of the environment variable `varname` if it exists, or `value` if it doesn’t. `value` defaults to `None`.

Availability: most flavors of Unix, Windows.

```python
os.putenv(varname, value)
```
Set the environment variable named `varname` to the string `value`. Such changes to the environment affect subprocesses started with `os.system()`, `popen()` or `fork()` and `execv()`.

Availability: most flavors of Unix, Windows.

**Note:** On some platforms, including FreeBSD and Mac OS X, setting `environ` may cause memory leaks. Refer to the system documentation for `putenv`.

When `putenv()` is supported, assignments to items in `os.environ` are automatically translated into corresponding calls to `putenv()`; however, calls to `putenv()` don’t update `os.environ`, so it is actually preferable to assign to items of `os.environ`.

```python
os.setegid(egid)
```
Set the current process's effective group id.

Availability: Unix.

`os.seteuid(euid)`
Set the current process's effective user id.

Availability: Unix.

`os.setgid(gid)`
Set the current process' group id.

Availability: Unix.

`os.setgroups(groups)`
Set the list of supplemental group ids associated with the current process to `groups`. `groups` must be a sequence, and each element must be an integer identifying a group. This operation is typically available only to the superuser.

Availability: Unix.

*New in version 2.2.*

**Note:** On Mac OS X, the length of `groups` may not exceed the system-defined maximum number of effective group ids, typically 16. See the documentation for `getgroups()` for cases where it may not return the same group list set by calling `setgroups()`.

`os.setpgrp()`
Call the system call `setpgrp()` or `setpgrp(θ, θ)()` depending on which version is implemented (if any). See the Unix manual for the semantics.

Availability: Unix.
**os.setpgid**(*pid, pgrp*)
Call the system call *setpgid()* to set the process group id of the process with id *pid* to the process group with id *pgrp*. See the Unix manual for the semantics.

Availability: Unix.

**os.setregid**(*rgid, egid*)
Set the current process’s real and effective group ids.

Availability: Unix.

**os.setresgid**(*rgid, egid, sgid*)
Set the current process’s real, effective, and saved group ids.

Availability: Unix.

*New in version 2.7.*

**os.setresuid**(*ruid, euid, suid*)
Set the current process’s real, effective, and saved user ids.

Availability: Unix.

*New in version 2.7.*

**os.setreuid**(*ruid, euid*)
Set the current process’s real and effective user ids.

Availability: Unix.

**os.getsid**(*pid*)
Call the system call *getsid()* . See the Unix manual for the semantics.

Availability: Unix.
New in version 2.4.

**os.setsid()**
Call the system call `setsid()`. See the Unix manual for the semantics.

*Availability: Unix.*

**os.setuid(uid)**
Set the current process's user id.

*Availability: Unix.*

**os.strerror(code)**
Return the error message corresponding to the error code in `code`. On platforms where `strerror()` returns `NULL` when given an unknown error number, `ValueError` is raised.

*Availability: Unix, Windows.*

**os.umask(mask)**
Set the current numeric umask and return the previous umask.

*Availability: Unix, Windows.*

**os.uname()**
Return a 5-tuple containing information identifying the current operating system. The tuple contains 5 strings: `(sysname, nodename, release, version, machine)`. Some systems truncate the nodename to 8 characters or to the leading component; a better way to get the hostname is `socket.gethostname()` or even `socket.gethostbyaddr(socket.gethostname())`.

*Availability: recent flavors of Unix.*

**os.unsetenv(varname)**
Unset (delete) the environment variable named varname. Such changes to the environment affect subprocesses started with \texttt{os.system()}, \texttt{popen()} or \texttt{fork()} and \texttt{execv()}.

When \texttt{unsetenv()} is supported, deletion of items in \texttt{os.environ} is automatically translated into a corresponding call to \texttt{unsetenv()}; however, calls to \texttt{unsetenv()} don’t update \texttt{os.environ}, so it is actually preferable to delete items of \texttt{os.environ}.

Availability: most flavors of Unix, Windows.
15.1.2. File Object Creation

These functions create new file objects. (See also `open()`.)

```python
os.fdopen(fd[, mode[, bufsize]])
```

Return an open file object connected to the file descriptor `fd`. The `mode` and `bufsize` arguments have the same meaning as the corresponding arguments to the built-in `open()` function. If `fdopen()` raises an exception, it leaves `fd` untouched (unclosed).

**Availability:** Unix, Windows.

*Changed in version 2.3:* When specified, the `mode` argument must now start with one of the letters 'r', 'w', or 'a', otherwise a `ValueError` is raised.

*Changed in version 2.5:* On Unix, when the `mode` argument starts with 'a', the O_APPEND flag is set on the file descriptor (which the `fdopen()` implementation already does on most platforms).

```python
os.popen(command[, mode[, bufsize]])
```

Open a pipe to or from `command`. The return value is an open file object connected to the pipe, which can be read or written depending on whether `mode` is 'r' (default) or 'w'. The `bufsize` argument has the same meaning as the corresponding argument to the built-in `open()` function. The exit status of the command (encoded in the format specified for `wait()`) is available as the return value of the `close()` method of the file object, except that when the exit status is zero (termination without errors), `None` is returned.
Availability: Unix, Windows.

**Deprecated since version 2.6:** This function is obsolete. Use the `subprocess` module. Check especially the Replacing Older Functions with the `subprocess` Module section.

**Changed in version 2.0:** This function worked unreliably under Windows in earlier versions of Python. This was due to the use of the `_popen()` function from the libraries provided with Windows. Newer versions of Python do not use the broken implementation from the Windows libraries.

```python
os.tmpfile()
```

Return a new file object opened in update mode (`w+b`). The file has no directory entries associated with it and will be automatically deleted once there are no file descriptors for the file.

Availability: Unix, Windows.

There are a number of different `popen*()` functions that provide slightly different ways to create subprocesses.

**Deprecated since version 2.6:** All of the `popen*()` functions are obsolete. Use the `subprocess` module.

For each of the `popen*()` variants, if `bufsize` is specified, it specifies the buffer size for the I/O pipes. `mode`, if provided, should be the string `'b'` or `'t'`; on Windows this is needed to determine whether the file objects should be opened in binary or text mode. The default value for `mode` is `'t'`.

Also, for each of these variants, on Unix, `cmd` may be a sequence, in which case arguments will be passed directly to the program without
shell intervention (as with \texttt{os.spawnv()}). If \textit{cmd} is a string it will be passed to the shell (as with \texttt{os.system()}).

These methods do not make it possible to retrieve the exit status from the child processes. The only way to control the input and output streams and also retrieve the return codes is to use the \texttt{subprocess} module; these are only available on Unix.

For a discussion of possible deadlock conditions related to the use of these functions, see Flow Control Issues.

\texttt{os.popen2(cmd[, mode[, bufsize]])}

Execute \textit{cmd} as a sub-process and return the file objects \texttt{(child_stdin, child_stdout)}.

\textit{Deprecated since version 2.6:} This function is obsolete. Use the \texttt{subprocess} module. Check especially the Replacing Older Functions with the subprocess Module section.

Availability: Unix, Windows.

\textit{New in version 2.0.}

\texttt{os.popen3(cmd[, mode[, bufsize]])}

Execute \textit{cmd} as a sub-process and return the file objects \texttt{(child_stdin, child_stdout, child_stderr)}.

\textit{Deprecated since version 2.6:} This function is obsolete. Use the \texttt{subprocess} module. Check especially the Replacing Older Functions with the subprocess Module section.

Availability: Unix, Windows.

\textit{New in version 2.0.}
`os.popen4(cmd[, mode[, bufsize]])`

Execute `cmd` as a sub-process and return the file objects
`(child_stdin, child_stdout_and_stderr).

**Deprecated since version 2.6:** This function is obsolete. Use
the `subprocess` module. Check especially the Replacing Older
Functions with the subprocess Module section.

Availability: Unix, Windows.

*New in version 2.0.*

(Note that `child_stdin`, `child_stdout`, and `child_stderr` are named
from the point of view of the child process, so `child_stdin` is the
child's standard input.)

This functionality is also available in the `popen2` module using
functions of the same names, but the return values of those
functions have a different order.)
15.1.3. File Descriptor Operations

These functions operate on I/O streams referenced using file descriptors.

File descriptors are small integers corresponding to a file that has been opened by the current process. For example, standard input is usually file descriptor 0, standard output is 1, and standard error is 2. Further files opened by a process will then be assigned 3, 4, 5, and so forth. The name “file descriptor” is slightly deceptive; on Unix platforms, sockets and pipes are also referenced by file descriptors.

The `fileno()` method can be used to obtain the file descriptor associated with a file object when required. Note that using the file descriptor directly will bypass the file object methods, ignoring aspects such as internal buffering of data.

```python
os.close(fd)
```

Close file descriptor `fd`.

Availability: Unix, Windows.

**Note:** This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `pipe()`. To close a “file object” returned by the built-in function `open()` or by `popen()` or `fdopen()`, use its `close()` method.

```python
os.closerange(fd_low, fd_high)
```

Close all file descriptors from `fd_low` (inclusive) to `fd_high` (exclusive), ignoring errors. Equivalent to:

```python
for fd in xrange(fd_low, fd_high):
    try:
        os.close(fd)
```
except OSError:
    pass

Availability: Unix, Windows.

*New in version 2.6.*

**os. dup(fd)**

Return a duplicate of file descriptor *fd*.

Availability: Unix, Windows.

**os. dup2(fd, fd2)**

Duplicate file descriptor *fd* to *fd2*, closing the latter first if necessary.

Availability: Unix, Windows.

**os. fchmod(fd, mode)**

Change the mode of the file given by *fd* to the numeric *mode*. See the docs for `chmod()` for possible values of *mode*.

Availability: Unix.

*New in version 2.6.*

**os. fchown(fd, uid, gid)**

Change the owner and group id of the file given by *fd* to the numeric *uid* and *gid*. To leave one of the ids unchanged, set it to -1.

Availability: Unix.

*New in version 2.6.*

**os. fdatasync(fd)**

Force write of file with filedescriptor *fd* to disk. Does not force
update of metadata.

Availability: Unix.

**Note:** This function is not available on MacOS.

```python
os.pathconf(fd, name)
```

Return system configuration information relevant to an open file. *name* specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the `pathconf_names` dictionary. For configuration variables not included in that mapping, passing an integer for *name* is also accepted.

If *name* is a string and is not known, `ValueError` is raised. If a specific value for *name* is not supported by the host system, even if it is included in `pathconf_names`, an `OSError` is raised with `errno.EINVAL` for the error number.

Availability: Unix.

```python
os.fstat(fd)
```

Return status for file descriptor *fd*, like `stat()`.

Availability: Unix, Windows.

```python
os.fstatvfs(fd)
```

Return information about the filesystem containing the file associated with file descriptor *fd*, like `statvfs()`.

Availability: Unix.
**os.``fsync(fd)``**

Force write of file with filedescriptor `fd` to disk. On Unix, this calls the native `fsync()` function; on Windows, the MS `_commit()` function.

If you're starting with a Python file object `f`, first do `f.flush()`, and then do `os.fsync(f.fileno())`, to ensure that all internal buffers associated with `f` are written to disk.

*Availability*: Unix, and Windows starting in 2.2.3.

**os.``ftruncate(fd, length)``**

Truncate the file corresponding to file descriptor `fd`, so that it is at most `length` bytes in size.

*Availability*: Unix.

**os.``isatty(fd)``**

Return `True` if the file descriptor `fd` is open and connected to a tty(-like) device, else `False`.

**os.``lseek(fd, pos, how)``**

Set the current position of file descriptor `fd` to position `pos`, modified by `how`: `SEEK_SET` or `0` to set the position relative to the beginning of the file; `SEEK_CUR` or `1` to set it relative to the current position; `SEEK_END` or `2` to set it relative to the end of the file. Return the new cursor position in bytes, starting from the beginning.

*Availability*: Unix, Windows.

**os.``SEEK_SET``**

**os.``SEEK_CUR``**

**os.``SEEK_END``**

Parameters to the `lseek()` function. Their values are 0, 1, and 2,
respectively.

Availability: Windows, Unix.

*New in version 2.5.*

```python
os.open(file, flags[, mode])
```

Open the file `file` and set various flags according to `flags` and possibly its mode according to `mode`. The default `mode` is `0777` (octal), and the current umask value is first masked out. Return the file descriptor for the newly opened file.

For a description of the flag and mode values, see the C run-time documentation; flag constants (like `O_RDONLY` and `O_WRONLY`) are defined in this module too (see `open()` flag constants). In particular, on Windows adding `O_BINARY` is needed to open files in binary mode.

Availability: Unix, Windows.

**Note:** This function is intended for low-level I/O. For normal usage, use the built-in function `open()`, which returns a “file object” with `read()` and `write()` methods (and many more). To wrap a file descriptor in a “file object”, use `fdopen()`.

```python
os.openpty()
```

Open a new pseudo-terminal pair. Return a pair of file descriptors `(master, slave)` for the pty and thetty, respectively. For a (slightly) more portable approach, use the `pty` module.

Availability: some flavors of Unix.

```python
os.pipe()
```

Create a pipe. Return a pair of file descriptors `(r, w)` usable for
reading and writing, respectively.

Availability: Unix, Windows.

**os. read(fd, n)**
Read at most n bytes from file descriptor fd. Return a string containing the bytes read. If the end of the file referred to by fd has been reached, an empty string is returned.

Availability: Unix, Windows.

**Note:** This function is intended for low-level I/O and must be applied to a file descriptor as returned by **os.open()** or **pipe()**. To read a “file object” returned by the built-in function **open()** or by **popen()** or **fdopen()**, or **sys.stdin**, use its **read()** or **readline()** methods.

**os. tcgetpgrp(fd)**
Return the process group associated with the terminal given by fd (an open file descriptor as returned by **os.open()**).

Availability: Unix.

**os. tcsetpgrp(fd, pg)**
Set the process group associated with the terminal given by fd (an open file descriptor as returned by **os.open()**) to pg.

Availability: Unix.

**os. ttyname(fd)**
Return a string which specifies the terminal device associated with file descriptor fd. If fd is not associated with a terminal device, an exception is raised.

Availability: Unix.
os.write(fd, str)

Write the string \textit{str} to file descriptor \textit{fd}. Return the number of bytes actually written.

\textbf{Availability:} Unix, Windows.

\textbf{Note:} This function is intended for low-level I/O and must be applied to a file descriptor as returned by \texttt{os.open()} or \texttt{pipe()}. To write a “file object” returned by the built-in function \texttt{open()} or by \texttt{open()} or \texttt{fdopen()}, or \texttt{sys.stdout} or \texttt{sys.stderr}, use its \texttt{write()} method.

\subsection*{15.1.3.1. open() flag constants}

The following constants are options for the \texttt{flags} parameter to the \texttt{open()} function. They can be combined using the bitwise OR operator \texttt{\textbar}. Some of them are not available on all platforms. For descriptions of their availability and use, consult the \texttt{open(2)} manual page on Unix or the MSDN on Windows.

\begin{verbatim}

os.O_RDONLY
os.O_WRONLY
os.O_RDWR
os.O_APPEND
os.O_CREAT
os.O_EXCL
os.O_TRUNC

The above constants are available on Unix and Windows.

os.O_DSYNC
os.O_RSYNC
os.O_SYNC
os.O_NDELAY
os.O_NONBLOCK
os.O_NOCTTY

The above constants are only available on Unix.
\end{verbatim}
os. O_BINARY
os. O_NOINHERIT
os. O_SHORT_LIVED
os. O_TEMPORARY
os. O_RANDOM
os. O_SEQUENTIAL
os. O_TEXT

The above constants are only available on Windows.

os. O_ASYNC
os. O_DIRECT
os. O_DIRECTORY
os. O_NOFOLLOW
os. O_NOATIME
os. O_SHLOCK
os. O_EXLOCK

The above constants are extensions and not present if they are not defined by the C library.
15.1.4. Files and Directories

`os.access(path, mode)`

Use the real uid/gid to test for access to `path`. Note that most operations will use the effective uid/gid, therefore this routine can be used in a suid/sgid environment to test if the invoking user has the specified access to `path`. `mode` should be `F_OK` to test the existence of `path`, or it can be the inclusive OR of one or more of `R_OK`, `W_OK`, and `X_OK` to test permissions. Return `True` if access is allowed, `False` if not. See the Unix man page `access(2)` for more information.

Availability: Unix, Windows.

**Note:** Using `access()` to check if a user is authorized to e.g. open a file before actually doing so using `open()` creates a security hole, because the user might exploit the short time interval between checking and opening the file to manipulate it. It's preferable to use EAFP techniques. For example:

```python
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()
return "some default data"
```

is better written as:

```python
try:
    fp = open("myfile")
except IOError as e:
    if e.errno == errno.EACCESS:
        return "some default data"
        # Not a permission error.
        raise
else:
    with fp:
        return fp.read()
```
**Note:** I/O operations may fail even when `access()` indicates that they would succeed, particularly for operations on network filesystems which may have permissions semantics beyond the usual POSIX permission-bit model.

`os.F_OK`
Value to pass as the `mode` parameter of `access()` to test the existence of `path`.

`os.R_OK`
Value to include in the `mode` parameter of `access()` to test the readability of `path`.

`os.W_OK`
Value to include in the `mode` parameter of `access()` to test the writability of `path`.

`os.X_OK`
Value to include in the `mode` parameter of `access()` to determine if `path` can be executed.

`os.chdir(path)`
Change the current working directory to `path`.

*Availability:* Unix, Windows.

`os.fchdir(fd)`
Change the current working directory to the directory represented by the file descriptor `fd`. The descriptor must refer to an opened directory, not an open file.

*Availability:* Unix.

*New in version 2.3.*
os.getcwd()
Return a string representing the current working directory.

Availability: Unix, Windows.

os.getcwdu()
Return a Unicode object representing the current working directory.

Availability: Unix, Windows.

*New in version 2.3.*

os.chflags(path, flags)
Set the flags of *path* to the numeric *flags*. *flags* may take a combination (bitwise OR) of the following values (as defined in the *stat* module):

- `stat.UF_NODUMP`
- `stat.UF_IMMUTABLE`
- `stat.UF_APPEND`
- `stat.UF_OPAQUE`
- `stat.UF_NOUNLINK`
- `stat.UF_COMPRESSED`
- `stat.UF_HIDDEN`
- `stat.SF_ARCHIVED`
- `stat.SF_IMMUTABLE`
- `stat.SF_APPEND`
- `stat.SF_NOUNLINK`
- `stat.SF_SNAPSHOT`

Availability: Unix.

*New in version 2.6.*
**os.chroot**(path)

Change the root directory of the current process to *path*. 
Availability: Unix.

*New in version 2.2.*

**os.chmod**(path, mode)

Change the mode of *path* to the numeric *mode*. *mode* may take one of the following values (as defined in the stat module) or bitwise ORed combinations of them:

- stat.S_ISUID
- stat.S_ISGID
- stat.S_ENFMT
- stat.S_ISVTX
- stat.S_IREAD
- stat.S_IWRITE
- stat.S_IEXEC
- stat.S_IRWXU
- stat.S_IRUSR
- stat.S_IWUSR
- stat.S_IXUSR
- stat.S_IRWXG
- stat.S_IRGRP
- stat.S_IWGRP
- stat.S_IXGRP
- stat.S_IRWXO
- stat.S_IROTH
- stat.S_IWOTH
- stat.S_IXOTH

Availability: Unix, Windows.
**Note:** Although Windows supports `chmod()`, you can only set the file's read-only flag with it (via the `stat.S_IWRITE` and `stat.S_IREAD` constants or a corresponding integer value). All other bits are ignored.

`os.chown(path, uid, gid)`  
Change the owner and group id of `path` to the numeric `uid` and `gid`. To leave one of the ids unchanged, set it to -1.  
Availability: Unix.

`os.lchflags(path, flags)`  
Set the flags of `path` to the numeric `flags`, like `chflags()`, but do not follow symbolic links.  
Availability: Unix.  
*New in version 2.6.*

`os.lchmod(path, mode)`  
Change the mode of `path` to the numeric `mode`. If path is a symlink, this affects the symlink rather than the target. See the docs for `chmod()` for possible values of `mode`.  
Availability: Unix.  
*New in version 2.6.*

`os.lchown(path, uid, gid)`  
Change the owner and group id of `path` to the numeric `uid` and `gid`. This function will not follow symbolic links.  
Availability: Unix.  
*New in version 2.3.*
os. `link(source, link_name)`

Create a hard link pointing to `source` named `link_name`.

Availability: Unix.

os. `listdir(path)`

Return a list containing the names of the entries in the directory given by `path`. The list is in arbitrary order. It does not include the special entries `.' and `..' even if they are present in the directory.

Availability: Unix, Windows.

*Changed in version 2.3:* On Windows NT/2k/XP and Unix, if `path` is a Unicode object, the result will be a list of Unicode objects. Undecodable filenames will still be returned as string objects.

os. `lstat(path)`

Perform the equivalent of an `lstat()` system call on the given path. Similar to `stat()`, but does not follow symbolic links. On platforms that do not support symbolic links, this is an alias for `stat()`.

os. `mkfifo(path[, mode])`

Create a FIFO (a named pipe) named `path` with numeric mode `mode`. The default `mode` is 0666 (octal). The current umask value is first masked out from the mode.

Availability: Unix.

FIFOs are pipes that can be accessed like regular files. FIFOs exist until they are deleted (for example with `os.unlink()`). Generally, FIFOs are used as rendezvous between “client” and “server” type processes: the server opens the FIFO for reading, and the client opens it for writing. Note that `mkfifo()` doesn’t
open the FIFO — it just creates the rendezvous point.

```python
os.mknod(filename[, mode=0600[, device=0]])
```

Create a filesystem node (file, device special file or named pipe) named `filename`. `mode` specifies both the permissions to use and the type of node to be created, being combined (bitwise OR) with one of `stat.S_IFREG`, `stat.S_IFCHR`, `stat.S_IFBLK`, and `stat.S_IFIFO` (those constants are available in `stat`). For `stat.S_IFCHR` and `stat.S_IFBLK`, `device` defines the newly created device special file (probably using `os.makedev()`), otherwise it is ignored.

*New in version 2.3.*

```python
os.major(device)
```

Extract the device major number from a raw device number (usually the `st_dev` or `st_rdev` field from `stat`).

*New in version 2.3.*

```python
os.minor(device)
```

Extract the device minor number from a raw device number (usually the `st_dev` or `st_rdev` field from `stat`).

*New in version 2.3.*

```python
os.makedev(major, minor)
```

Compose a raw device number from the major and minor device numbers.

*New in version 2.3.*

```python
os.mkdir(path[, mode])
```

Create a directory named `path` with numeric mode `mode`. The default `mode` is `0777` (octal). If the directory already exists,
**OSError** is raised.

On some systems, *mode* is ignored. Where it is used, the current umask value is first masked out. If bits other than the last 9 (i.e. the last 3 digits of the octal representation of the *mode*) are set, their meaning is platform-dependent. On some platforms, they are ignored and you should call `chmod()` explicitly to set them.

It is also possible to create temporary directories; see the `tempfile` module's `tempfile.mkdtemp()` function.

**Availability:** Unix, Windows.

```python
os.makedirs(path[, mode])
```

Recursive directory creation function. Like `mkdir()`, but makes all intermediate-level directories needed to contain the leaf directory. Raises an `error` exception if the leaf directory already exists or cannot be created. The default *mode* is `0777` (octal).

The *mode* parameter is passed to `mkdir()`; see the `mkdir()` description for how it is interpreted.

**Note:** `makedirs()` will become confused if the path elements to create include `os.pardir`.

*New in version 1.5.2.*

*Changed in version 2.3:* This function now handles UNC paths correctly.

```python
os.pathconf(path, name)
```

Return system configuration information relevant to a named file. *name* specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix
98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the `pathconf_names` dictionary. For configuration variables not included in that mapping, passing an integer for `name` is also accepted.

If `name` is a string and is not known, `ValueError` is raised. If a specific value for `name` is not supported by the host system, even if it is included in `pathconf_names`, an `OSError` is raised with `errno.EINVAL` for the error number.

Availability: Unix.

**os.pathconf_names**

Dictionary mapping names accepted by `pathconf()` and `fpathconf()` to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system. Availability: Unix.

**os.readlink(path)**

Return a string representing the path to which the symbolic link points. The result may be either an absolute or relative pathname; if it is relative, it may be converted to an absolute pathname using `os.path.join(os.path.dirname(path), result)`.

*Changed in version 2.6:* If the `path` is a Unicode object the result will also be a Unicode object.

Availability: Unix.

**os.remove(path)**

Remove (delete) the file `path`. If `path` is a directory, `OSError` is raised; see `rmdir()` below to remove a directory. This is identical to the `unlink()` function documented below. On Windows, attempting to remove a file that is in use causes an exception to
be raised; on Unix, the directory entry is removed but the storage allocated to the file is not made available until the original file is no longer in use.

Availability: Unix, Windows.

**os.removedirs(path)**
Remove directories recursively. Works like `rmdir()` except that, if the leaf directory is successfully removed, `removedirs()` tries to successively remove every parent directory mentioned in `path` until an error is raised (which is ignored, because it generally means that a parent directory is not empty). For example, `os.removedirs('foo/bar/baz')` will first remove the directory 'foo/bar/baz', and then remove 'foo/bar' and 'foo' if they are empty. Raises `OSError` if the leaf directory could not be successfully removed.

*New in version 1.5.2.*

**os.rename(src, dst)**
Rename the file or directory `src` to `dst`. If `dst` is a directory, `OSError` will be raised. On Unix, if `dst` exists and is a file, it will be replaced silently if the user has permission. The operation may fail on some Unix flavors if `src` and `dst` are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement). On Windows, if `dst` already exists, `OSError` will be raised even if it is a file; there may be no way to implement an atomic rename when `dst` names an existing file.

Availability: Unix, Windows.

**os.renames(old, new)**
Recursive directory or file renaming function. Works like
rename(), except creation of any intermediate directories needed to make the new pathname good is attempted first. After the rename, directories corresponding to rightmost path segments of the old name will be pruned away using removedirs().

*New in version 1.5.2.*

**Note:** This function can fail with the new directory structure made if you lack permissions needed to remove the leaf directory or file.

```python
os.rmdir(path)
```

Remove (delete) the directory path. Only works when the directory is empty, otherwise, OSError is raised. In order to remove whole directory trees, shutil.rmtree() can be used.

Availability: Unix, Windows.

```python
os.stat(path)
```

Perform the equivalent of a stat() system call on the given path. (This function follows symlinks; to stat a symlink use lstat().)

The return value is an object whose attributes correspond to the members of the stat structure, namely:

- st_mode - protection bits,
- st_ino - inode number,
- st_dev - device,
- st_nlink - number of hard links,
- st_uid - user id of owner,
- st_gid - group id of owner,
- st_size - size of file, in bytes,
- st_atime - time of most recent access,
- **st_mtime** - time of most recent content modification,
- **st_ctime** - platform dependent; time of most recent metadata change on Unix, or the time of creation on Windows)

*Changed in version 2.3:* If `stat_float_times()` returns `True`, the time values are floats, measuring seconds. Fractions of a second may be reported if the system supports that. See `stat_float_times()` for further discussion.

On some Unix systems (such as Linux), the following attributes may also be available:

- **st_blocks** - number of 512-byte blocks allocated for file
- **st_blksize** - filesystem blocksize for efficient file system I/O
- **st_rdev** - type of device if an inode device
- **st_flags** - user defined flags for file

On other Unix systems (such as FreeBSD), the following attributes may be available (but may be only filled out if root tries to use them):

- **st_gen** - file generation number
- **st_birthtime** - time of file creation

On RISCOS systems, the following attributes are also available:

- **st_fstype** (file type)
- **st_attrs** (attributes)
- **st_obtype** (object type).

**Note:** The exact meaning and resolution of the **st_atime**, **st_mtime**, and **st_ctime** attributes depend on the operating system and the file system. For example, on Windows systems using the FAT or FAT32 file systems, **st_mtime** has 2-second resolution, and **st_atime** has only 1-day resolution. See your
For backward compatibility, the return value of \texttt{stat()} is also accessible as a tuple of at least 10 integers giving the most important (and portable) members of the \texttt{stat} structure, in the order \texttt{st\_mode}, \texttt{st\_ino}, \texttt{st\_dev}, \texttt{st\_nlink}, \texttt{st\_uid}, \texttt{st\_gid}, \texttt{st\_size}, \texttt{st\_atime}, \texttt{st\_mtime}, \texttt{st\_ctime}. More items may be added at the end by some implementations.

The standard module \texttt{stat} defines functions and constants that are useful for extracting information from a \texttt{stat} structure. (On Windows, some items are filled with dummy values.)

Example:

```python
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
(33188, 422511, 769, 1, 1032, 100, 926, 1105022698,1105022732
>>> statinfo.st_size
926
```

Availability: Unix, Windows.

\textit{Changed in version 2.2:} Added access to values as attributes of the returned object.

\textit{Changed in version 2.5:} Added \texttt{st\_gen} and \texttt{st\_birthtime}.

\texttt{os.stat\_float\_times([newvalue])}

Determine whether \texttt{stat\_result} represents time stamps as float objects. If \texttt{newvalue} is \texttt{True}, future calls to \texttt{stat()} return floats, if it is \texttt{False}, future calls return ints. If \texttt{newvalue} is omitted, return the current setting.
For compatibility with older Python versions, accessing \texttt{stat_result} as a tuple always returns integers.

\textit{Changed in version 2.5:} Python now returns float values by default. Applications which do not work correctly with floating point time stamps can use this function to restore the old behaviour.

The resolution of the timestamps (that is the smallest possible fraction) depends on the system. Some systems only support second resolution; on these systems, the fraction will always be zero.

It is recommended that this setting is only changed at program startup time in the \texttt{__main__} module; libraries should never change this setting. If an application uses a library that works incorrectly if floating point time stamps are processed, this application should turn the feature off until the library has been corrected.

\texttt{os.statvfs(path)}

Perform a \texttt{statvfs()} system call on the given path. The return value is an object whose attributes describe the filesystem on the given path, and correspond to the members of the \texttt{statvfs} structure, namely: \texttt{f_bsize, f_frsz, f_blocks, f_bfree, f_bavail, f_files, f_ffree, f_favail, f_flag, f_namemax}.

For backward compatibility, the return value is also accessible as a tuple whose values correspond to the attributes, in the order given above. The standard module \texttt{statvfs} defines constants that are useful for extracting information from a \texttt{statvfs} structure when accessing it as a sequence; this remains useful when writing code that needs to work with versions of Python that don’t support accessing the fields as attributes.
Availability: Unix.

*Changed in version 2.2:* Added access to values as attributes of the returned object.

```python
os.symlink(source, link_name)
```
Create a symbolic link pointing to `source` named `link_name`.

Availability: Unix.

```python
os.tempnam([dir[, prefix]])
```
Return a unique path name that is reasonable for creating a temporary file. This will be an absolute path that names a potential directory entry in the directory `dir` or a common location for temporary files if `dir` is omitted or `None`. If given and not `None`, `prefix` is used to provide a short prefix to the filename. Applications are responsible for properly creating and managing files created using paths returned by `tempnam()`; no automatic cleanup is provided. On Unix, the environment variable `TMPDIR` overrides `dir`, while on Windows `TMP` is used. The specific behavior of this function depends on the C library implementation; some aspects are underspecified in system documentation.

**Warning:** Use of `tempnam()` is vulnerable to symlink attacks; consider using `tmpfile()` (section File Object Creation) instead.

Availability: Unix, Windows.

```python
os.tmpnam()
```
Return a unique path name that is reasonable for creating a temporary file. This will be an absolute path that names a potential directory entry in a common location for temporary files. Applications are responsible for properly creating and managing
files created using paths returned by `tmpnam()`; no automatic cleanup is provided.

**Warning:** Use of `tmpnam()` is vulnerable to symlink attacks; consider using `tmpfile()` (section File Object Creation) instead.

Availability: Unix, Windows. This function probably shouldn’t be used on Windows, though: Microsoft’s implementation of `tmpnam()` always creates a name in the root directory of the current drive, and that’s generally a poor location for a temp file (depending on privileges, you may not even be able to open a file using this name).

**os.** `TMP_MAX`

The maximum number of unique names that `tmpnam()` will generate before reusing names.

**os.** `unlink(path)`

Remove (delete) the file `path`. This is the same function as `remove()`; the `unlink()` name is its traditional Unix name.

Availability: Unix, Windows.

**os.** `utime(path, times)`

Set the access and modified times of the file specified by `path`. If `times` is `None`, then the file’s access and modified times are set to the current time. (The effect is similar to running the Unix program `touch` on the path.) Otherwise, `times` must be a 2-tuple of numbers, of the form `(atime, mtime)` which is used to set the access and modified times, respectively. Whether a directory can be given for `path` depends on whether the operating system implements directories as files (for example, Windows does not). Note that the exact times you set here may not be returned by a subsequent `stat()` call, depending on the resolution with which
your operating system records access and modification times; see \texttt{stat()}.

\textit{Changed in version 2.0:} Added support for \texttt{None} for \texttt{times}.

Availability: Unix, Windows.

\texttt{os.walk(top, topdown=True, onerror=None, followlinks=False)}

Generate the file names in a directory tree by walking the tree either top-down or bottom-up. For each directory in the tree rooted at directory \texttt{top} (including \texttt{top} itself), it yields a 3-tuple \texttt{(dirpath, dirnames, filenames)}.

\texttt{dirpath} is a string, the path to the directory. \texttt{dirnames} is a list of the names of the subdirectories in \texttt{dirpath} (excluding \texttt{'.'} and '\texttt{..'\}). \texttt{filenames} is a list of the names of the non-directory files in \texttt{dirpath}. Note that the names in the lists contain no path components. To get a full path (which begins with \texttt{top}) to a file or directory in \texttt{dirpath}, do \texttt{os.path.join(dirpath, name)}.

If optional argument \texttt{topdown} is \texttt{True} or not specified, the triple for a directory is generated before the triples for any of its subdirectories (directories are generated top-down). If \texttt{topdown} is \texttt{False}, the triple for a directory is generated after the triples for all of its subdirectories (directories are generated bottom-up). No matter the value of \texttt{topdown}, the list of subdirectories is retrieved before the tuples for the directory and its subdirectories are generated.

When \texttt{topdown} is \texttt{True}, the caller can modify the \texttt{dirnames} list in-place (perhaps using \texttt{del} or slice assignment), and \texttt{walk()} will only recurse into the subdirectories whose names remain in \texttt{dirnames}; this can be used to prune the search, impose a specific order of visiting, or even to inform \texttt{walk()} about
directories the caller creates or renames before it resumes \texttt{walk()} again. Modifying \texttt{dirnames} when \texttt{topdown} is \texttt{False} has no effect on the behavior of the walk, because in bottom-up mode the directories in \texttt{dirnames} are generated before \texttt{dirpath} itself is generated.

By default, errors from the \texttt{listdir()} call are ignored. If optional argument \texttt{onerror} is specified, it should be a function; it will be called with one argument, an \texttt{OSError} instance. It can report the error to continue with the walk, or raise the exception to abort the walk. Note that the filename is available as the \texttt{filename} attribute of the exception object.

By default, \texttt{walk()} will not walk down into symbolic links that resolve to directories. Set \texttt{followlinks} to \texttt{True} to visit directories pointed to by symlinks, on systems that support them.

\textit{New in version 2.6:} The \texttt{followlinks} parameter.

\textbf{Note:} Be aware that setting \texttt{followlinks} to \texttt{True} can lead to infinite recursion if a link points to a parent directory of itself. \texttt{walk()} does not keep track of the directories it visited already.

\textbf{Note:} If you pass a relative pathname, don’t change the current working directory between resumptions of \texttt{walk()}. \texttt{walk()} never changes the current directory, and assumes that its caller doesn’t either.

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn’t look under any CVS subdirectory:

```python
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
```
In the next example, walking the tree bottom-up is essential: `rmdir()` doesn't allow deleting a directory before the directory is empty:

```python
if 'CVS' in dirs:
    dirs.remove('CVS')  # don't visit CVS directories
```

```
# Delete everything reachable from the directory named in "top", assuming there are no symbolic links.
# CAUTION: This is dangerous! For example, if top == '/', # could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

_New in version 2.3._
15.1.5. Process Management

These functions may be used to create and manage processes.

The various exec* functions take a list of arguments for the new program loaded into the process. In each case, the first of these arguments is passed to the new program as its own name rather than as an argument a user may have typed on a command line. For the C programmer, this is the argv[0] passed to a program's main(). For example, os.execv('/bin/echo', ['foo', 'bar']) will only print bar on standard output; foo will seem to be ignored.

os.abort()

Generate a SIGABRT signal to the current process. On Unix, the default behavior is to produce a core dump; on Windows, the process immediately returns an exit code of 3. Be aware that calling this function will not call the Python signal handler registered for SIGABRT with signal.signal().

Availability: Unix, Windows.

os.exec1(path, arg0, arg1, ...)
os.execle(path, arg0, arg1, ..., env)
os.execlp(file, arg0, arg1, ...)
os.execlpe(file, arg0, arg1, ..., env)
os.execv(path, args)
os.execve(path, args, env)
os.execvp(file, args)
os.execvpe(file, args, env)

These functions all execute a new program, replacing the current process; they do not return. On Unix, the new executable is loaded into the current process, and will have the same process id as the caller. Errors will be reported as OSError exceptions.
The current process is replaced immediately. Open file objects and descriptors are not flushed, so if there may be data buffered on these open files, you should flush them using `sys.stdout.flush()` or `os.fsync()` before calling an `exec*` function.

The “l” and “v” variants of the `exec*` functions differ in how command-line arguments are passed. The “l” variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the `exec1*()` functions. The “v” variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the `args` parameter. In either case, the arguments to the child process should start with the name of the command being run, but this is not enforced.

The variants which include a “p” near the end (`exec1p()`, `exec1pe()`, `execvp()`, and `execvpe()`) will use the `PATH` environment variable to locate the program `file`. When the environment is being replaced (using one of the `exec*e` variants, discussed in the next paragraph), the new environment is used as the source of the `PATH` variable. The other variants, `exec1()`, `execle()`, `execv()`, and `execve()`, will not use the `PATH` variable to locate the executable; `path` must contain an appropriate absolute or relative path.

For `execle()`, `exec1pe()`, `execve()`, and `execvpe()` (note that these all end in “e”), the `env` parameter must be a mapping which is used to define the environment variables for the new process (these are used instead of the current process’ environment); the functions `exec1()`, `exec1p()`, `execv()`, and `execvp()` all cause the new process to inherit the environment of the current process.
Availability: Unix, Windows.

`os._exit(n)`
Exit the process with status `n`, without calling cleanup handlers, flushing stdio buffers, etc.

Availability: Unix, Windows.

**Note:** The standard way to exit is `sys.exit(n)`. `_exit()` should normally only be used in the child process after a `fork()`.

The following exit codes are defined and can be used with `_exit()`, although they are not required. These are typically used for system programs written in Python, such as a mail server’s external command delivery program.

**Note:** Some of these may not be available on all Unix platforms, since there is some variation. These constants are defined where they are defined by the underlying platform.

`os.EX_OK`
Exit code that means no error occurred.

Availability: Unix.

*New in version 2.3.*

`os.EX_USAGE`
Exit code that means the command was used incorrectly, such as when the wrong number of arguments are given.

Availability: Unix.

*New in version 2.3.*

`os.EX_DATAERR`
Exit code that means the input data was incorrect.

Availability: Unix.

*New in version 2.3.*

**os. EX_NOINPUT**

Exit code that means an input file did not exist or was not readable.

Availability: Unix.

*New in version 2.3.*

**os. EX_NOUSER**

Exit code that means a specified user did not exist.

Availability: Unix.

*New in version 2.3.*

**os. EX_NOHOST**

Exit code that means a specified host did not exist.

Availability: Unix.

*New in version 2.3.*

**os. EX_UNAVAILABLE**

Exit code that means that a required service is unavailable.

Availability: Unix.

*New in version 2.3.*

**os. EX_SOFTWARE**

Exit code that means an internal software error was detected.
Availability: Unix.

New in version 2.3.

```
\texttt{os.EX\_OSERR}
```
Exit code that means an operating system error was detected, such as the inability to fork or create a pipe.

Availability: Unix.

New in version 2.3.

```
\texttt{os.EX\_OSFILE}
```
Exit code that means some system file did not exist, could not be opened, or had some other kind of error.

Availability: Unix.

New in version 2.3.

```
\texttt{os.EX\_CANTCREATE}
```
Exit code that means a user specified output file could not be created.

Availability: Unix.

New in version 2.3.

```
\texttt{os.EX\_IOERR}
```
Exit code that means that an error occurred while doing I/O on some file.

Availability: Unix.

New in version 2.3.

```
\texttt{os.EX\_TEMPFAIL}
```
Exit code that means a temporary failure occurred. This indicates
something that may not really be an error, such as a network connection that couldn't be made during a retryable operation.

Availability: Unix.

*New in version 2.3.*

```
0s. EX_PROTOCOL
Exit code that means that a protocol exchange was illegal, invalid, or not understood.

Availability: Unix.

*New in version 2.3.*
```

```
0s. EX_NOPERM
Exit code that means that there were insufficient permissions to perform the operation (but not intended for file system problems).

Availability: Unix.

*New in version 2.3.*
```

```
0s. EX_CONFIG
Exit code that means that some kind of configuration error occurred.

Availability: Unix.

*New in version 2.3.*
```

```
0s. EX_NOTFOUND
Exit code that means something like “an entry was not found”.

Availability: Unix.

*New in version 2.3.*
```
os. **fork()**

Fork a child process. Return 0 in the child and the child's process id in the parent. If an error occurs **OSError** is raised.

Note that some platforms including FreeBSD <= 6.3, Cygwin and OS/2 EMX have known issues when using fork() from a thread.

**Warning:** See **ssl** for applications that use the SSL module with fork().

Availability: Unix.

os. **forkpty()**

Fork a child process, using a new pseudo-terminal as the child's controlling terminal. Return a pair of (pid, fd), where pid is 0 in the child, the new child's process id in the parent, and fd is the file descriptor of the master end of the pseudo-terminal. For a more portable approach, use the pty module. If an error occurs **OSError** is raised.

Availability: some flavors of Unix.

os. **kill(pid, sig)**

Send signal **sig** to the process **pid**. Constants for the specific signals available on the host platform are defined in the **signal** module.

Windows: The **signal.CTRL_C_EVENT** and **signal.CTRL_BREAK_EVENT** signals are special signals which can only be sent to console processes which share a common console window, e.g., some subprocesses. Any other value for **sig** will cause the process to be unconditionally killed by the TerminateProcess API, and the exit code will be set to **sig**. The Windows version of **kill()** additionally takes process handles to
be killed.

**New in version 2.7:** Windows support

```python
os.killpg(pgid, sig)
```
Send the signal `sig` to the process group `pgid`.

*Availability: Unix.*

**New in version 2.3.**

```python
os.nice(increment)
```
Add `increment` to the process’s “niceness”. Return the new niceness.

*Availability: Unix.*

```python
os.plock(op)
```
Lock program segments into memory. The value of `op` (defined in `<sys/lock.h>`) determines which segments are locked.

*Availability: Unix.*

```python
os.popen(...)
```
```python
os.popen2(...)
```
```python
os.popen3(...)
```
```python
os.popen4(...)
```
Run child processes, returning opened pipes for communications. These functions are described in section *File Object Creation.*

```python
os.spawnl(mode, path, ...)
```
```python
os.spawnle(mode, path, ..., env)
```
```python
os.spawnlp(mode, file, ...)
```
```python
os.spawnlpe(mode, file, ..., env)
```
```python
os.spawnv(mode, path, args)
```
```python
os.spawnve(mode, path, args, env)
```
```python
os.spawnvp(mode, file, args)
```
os. `spawnvpe(mode, file, args, env)`

Execute the program `path` in a new process.

(Note that the `subprocess` module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using these functions. Check especially the Replacing Older Functions with the subprocess Module section.)

If `mode` is `P_NOWAIT`, this function returns the process id of the new process; if `mode` is `P_WAIT`, returns the process's exit code if it exits normally, or `-signal`, where `signal` is the signal that killed the process. On Windows, the process id will actually be the process handle, so can be used with the `waitpid()` function.

The “l” and “v” variants of the `spawn*` functions differ in how command-line arguments are passed. The “l” variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the `spawnl*()` functions. The “v” variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the `args` parameter. In either case, the arguments to the child process must start with the name of the command being run.

The variants which include a second “p” near the end (`spawnlp()`, `spawnlpe()`, `spawnvp()`, and `spawnvpe()`) will use the `PATH` environment variable to locate the program `file`. When the environment is being replaced (using one of the `spawn*e` variants, discussed in the next paragraph), the new environment is used as the source of the `PATH` variable. The other variants, `spawnl()`, `spawnle()`, `spawnv()`, and `spawnve()`, will not use the `PATH` variable to locate the executable; `path` must contain an appropriate absolute or relative path.
For `spawnle()`, `spawnlpe()`, `spawnve()`, and `spawnvpe()` (note that these all end in “e”), the `env` parameter must be a mapping which is used to define the environment variables for the new process (they are used instead of the current process’ environment); the functions `spawnl()`, `spawnlp()`, `spawnv()`, and `spawnvp()` all cause the new process to inherit the environment of the current process. Note that keys and values in the `env` dictionary must be strings; invalid keys or values will cause the function to fail, with a return value of 127.

As an example, the following calls to `spawnlp()` and `spawnvpe()` are equivalent:

```python
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

Availability: Unix, Windows. `spawnlp()`, `spawnlpe()`, `spawnvp()` and `spawnvpe()` are not available on Windows. `spawnle()` and `spawnve()` are not thread-safe on Windows; we advise you to use the `subprocess` module instead.

*New in version 1.6.*

- `os.P_NOWAIT`
- `os.P_NOWAITO`

Possible values for the `mode` parameter to the `spawn*` family of functions. If either of these values is given, the `spawn*()` functions will return as soon as the new process has been created, with the process id as the return value.

Availability: Unix, Windows.
New in version 1.6.

**os\_P\_WAIT**
Possible value for the *mode* parameter to the *spawn* family of functions. If this is given as *mode*, the *spawn*() functions will not return until the new process has run to completion and will return the exit code of the process the run is successful, or `-signal` if a signal kills the process.

Availability: Unix, Windows.

New in version 1.6.

**os\_P\_DETACH**
**os\_P\_OVERLAY**
Possible values for the *mode* parameter to the *spawn* family of functions. These are less portable than those listed above. *P\_DETACH* is similar to *P\_NOWAIT*, but the new process is detached from the console of the calling process. If *P\_OVERLAY* is used, the current process will be replaced; the *spawn*() function will not return.

Availability: Windows.

New in version 1.6.

**os\_startfile(path[, operation])**
Start a file with its associated application.

When *operation* is not specified or `'open'`, this acts like double-clicking the file in Windows Explorer, or giving the file name as an argument to the *start* command from the interactive command shell: the file is opened with whatever application (if any) its extension is associated.
When another operation is given, it must be a “command verb” that specifies what should be done with the file. Common verbs documented by Microsoft are 'print' and 'edit' (to be used on files) as well as 'explore' and 'find' (to be used on directories).

startfile() returns as soon as the associated application is launched. There is no option to wait for the application to close, and no way to retrieve the application's exit status. The path parameter is relative to the current directory. If you want to use an absolute path, make sure the first character is not a slash ('/'); the underlying Win32 ShellExecute() function doesn't work if it is. Use the os.path.normpath() function to ensure that the path is properly encoded for Win32.

Availability: Windows.

New in version 2.0.

New in version 2.5: The operation parameter.

os.system(command)
Execute the command (a string) in a subshell. This is implemented by calling the Standard C function system(), and has the same limitations. Changes to sys.stdin, etc. are not reflected in the environment of the executed command.

On Unix, the return value is the exit status of the process encoded in the format specified for wait(). Note that POSIX does not specify the meaning of the return value of the C system() function, so the return value of the Python function is system-dependent.

On Windows, the return value is that returned by the system shell after running command, given by the Windows environment variable COMSPEC: on command.com systems (Windows 95, 98
and ME) this is always 0; on \texttt{cmd.exe} systems (Windows NT, 2000 and XP) this is the exit status of the command run; on systems using a non-native shell, consult your shell documentation.

The \texttt{subprocess} module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using this function. See the \texttt{Replacing Older Functions with the subprocess Module} section in the \texttt{subprocess} documentation for some helpful recipes.

Availability: Unix, Windows.

\texttt{os.times()}

Return a 5-tuple of floating point numbers indicating accumulated (processor or other) times, in seconds. The items are: user time, system time, children’s user time, children’s system time, and elapsed real time since a fixed point in the past, in that order. See the Unix manual page \texttt{times(2)} or the corresponding Windows Platform API documentation. On Windows, only the first two items are filled, the others are zero.

Availability: Unix, Windows

\texttt{os.wait()}

Wait for completion of a child process, and return a tuple containing its pid and exit status indication: a 16-bit number, whose low byte is the signal number that killed the process, and whose high byte is the exit status (if the signal number is zero); the high bit of the low byte is set if a core file was produced.

Availability: Unix.

\texttt{os.waitpid(pid, options)}

The details of this function differ on Unix and Windows.
On Unix: Wait for completion of a child process given by process id *pid*, and return a tuple containing its process id and exit status indication (encoded as for *wait()*). The semantics of the call are affected by the value of the integer *options*, which should be 0 for normal operation.

If *pid* is greater than 0, *waitpid()* requests status information for that specific process. If *pid* is 0, the request is for the status of any child in the process group of the current process. If *pid* is -1, the request pertains to any child of the current process. If *pid* is less than -1, status is requested for any process in the process group -pid (the absolute value of *pid*).

An *OSError* is raised with the value of errno when the syscall returns -1.

On Windows: Wait for completion of a process given by process handle *pid*, and return a tuple containing *pid*, and its exit status shifted left by 8 bits (shifting makes cross-platform use of the function easier). A *pid* less than or equal to 0 has no special meaning on Windows, and raises an exception. The value of integer *options* has no effect. *pid* can refer to any process whose id is known, not necessarily a child process. The spawn* functions called with *P_NOWAIT* return suitable process handles.

**os.wait3(options)**

Similar to *waitpid()*, except no process id argument is given and a 3-element tuple containing the child’s process id, exit status indication, and resource usage information is returned. Refer to *resource.getrusage()* for details on resource usage information. The option argument is the same as that provided to *waitpid()* and *wait4()*.

Availability: Unix.
**New in version 2.5.**

**os.wait4(pid, options)**

Similar to `waitpid()`, except a 3-element tuple, containing the child's process id, exit status indication, and resource usage information is returned. Refer to `resource.getusage()` for details on resource usage information. The arguments to `wait4()` are the same as those provided to `waitpid()`.

Availability: Unix.

**New in version 2.5.**

**os.WNOHANG**

The option for `waitpid()` to return immediately if no child process status is available immediately. The function returns `(0, 0)` in this case.

Availability: Unix.

**os.WCONTINUED**

This option causes child processes to be reported if they have been continued from a job control stop since their status was last reported.

Availability: Some Unix systems.

**New in version 2.3.**

**os.WUNTRACED**

This option causes child processes to be reported if they have been stopped but their current state has not been reported since they were stopped.

Availability: Unix.
New in version 2.3.

The following functions take a process status code as returned by `system()`, `wait()`, or `waitpid()` as a parameter. They may be used to determine the disposition of a process.

```
os.WCOREDUMP(status)
    Return True if a core dump was generated for the process, otherwise return False.

    Availability: Unix.
```

New in version 2.3.

```
os.WIFCONTINUED(status)
    Return True if the process has been continued from a job control stop, otherwise return False.

    Availability: Unix.
```

New in version 2.3.

```
os.WIFSTOPPED(status)
    Return True if the process has been stopped, otherwise return False.

    Availability: Unix.
```

```
os.WIFSIGNALED(status)
    Return True if the process exited due to a signal, otherwise return False.

    Availability: Unix.
```

```
os.WIFEXITED(status)
    Return True if the process exited using the `exit(2)` system call,
```
otherwise return False.

Availability: Unix.

os. **WEXITSTATUS**(status)

If **WIFEXITED**(status) is true, return the integer parameter to the **exit**(2) system call. Otherwise, the return value is meaningless.

Availability: Unix.

os. **WSTOPSIG**(status)

Return the signal which caused the process to stop.

Availability: Unix.

os. **WTERMSIG**(status)

Return the signal which caused the process to exit.

Availability: Unix.
15.1.6. Miscellaneous System Information

**os.confstr(name)**

Return string-valued system configuration values. *name* specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given as the keys of the *confstr_names* dictionary. For configuration variables not included in that mapping, passing an integer for *name* is also accepted.

If the configuration value specified by *name* isn't defined, *None* is returned.

If *name* is a string and is not known, *ValueError* is raised. If a specific value for *name* is not supported by the host system, even if it is included in *confstr_names*, an *OSError* is raised with *errno.EINVAL* for the error number.

Availability: Unix

**os.confstr_names**

Dictionary mapping names accepted by *confstr()* to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

Availability: Unix.

**os.getloadavg()**

Return the number of processes in the system run queue averaged over the last 1, 5, and 15 minutes or raises *OSError* if
the load average was unobtainable.

Availability: Unix.

New in version 2.3.

**os.sysconf(name)**

Return integer-valued system configuration values. If the configuration value specified by *name* isn't defined, \(-1\) is returned. The comments regarding the *name* parameter for *confstr()* apply here as well; the dictionary that provides information on the known names is given by *sysconf_names*.

Availability: Unix.

**os.sysconf_names**

Dictionary mapping names accepted by *sysconf()* to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

Availability: Unix.

The following data values are used to support path manipulation operations. These are defined for all platforms.

Higher-level operations on pathnames are defined in the *os.path* module.

**os.curdir**

The constant string used by the operating system to refer to the current directory. This is '.\.' for Windows and POSIX. Also available via *os.path*.

**os.pardir**

The constant string used by the operating system to refer to the
parent directory. This is '...' for Windows and POSIX. Also available via `os.path`.

**os.sep**
The character used by the operating system to separate pathname components. This is '/' for POSIX and '\\' for Windows. Note that knowing this is not sufficient to be able to parse or concatenate pathnames — use `os.path.split()` and `os.path.join()` — but it is occasionally useful. Also available via `os.path`.

**os.altsep**
An alternative character used by the operating system to separate pathname components, or `None` if only one separator character exists. This is set to '/' on Windows systems where `sep` is a backslash. Also available via `os.path`.

**os.extsep**
The character which separates the base filename from the extension; for example, the '.' in `os.py`. Also available via `os.path`.

*New in version 2.2.*

**os.pathsep**
The character conventionally used by the operating system to separate search path components (as in `PATH`), such as ':' for POSIX or ';' for Windows. Also available via `os.path`.

**os.defpath**
The default search path used by `exec*p*` and `spawn*p*` if the environment doesn’t have a `PATH` key. Also available via `os.path`.
os. **linesep**
The string used to separate (or, rather, terminate) lines on the current platform. This may be a single character, such as \n for POSIX, or multiple characters, for example, \r\n for Windows. Do not use os.linesep as a line terminator when writing files opened in text mode (the default); use a single \n instead, on all platforms.

os. **devnull**
The file path of the null device. For example: /dev/null for POSIX, nul for Windows. Also available via os.path.

*New in version 2.4.*
15.1.7. Miscellaneous Functions

`os.urandom(n)`

Return a string of `n` random bytes suitable for cryptographic use.

This function returns random bytes from an OS-specific randomness source. The returned data should be unpredictable enough for cryptographic applications, though its exact quality depends on the OS implementation. On a UNIX-like system this will query `/dev/urandom`, and on Windows it will use `CryptGenRandom()`. If a randomness source is not found, `NotImplementedError` will be raised.

For an easy-to-use interface to the random number generator provided by your platform, please see `random.SystemRandom`.

*New in version 2.4.*
15.2. `io` — Core tools for working with streams

*New in version 2.6.*

The `io` module provides the Python interfaces to stream handling. Under Python 2.x, this is proposed as an alternative to the built-in `file` object, but in Python 3.x it is the default interface to access files and streams.

**Note:** Since this module has been designed primarily for Python 3.x, you have to be aware that all uses of “bytes” in this document refer to the `str` type (of which `bytes` is an alias), and all uses of “text” refer to the `unicode` type. Furthermore, those two types are not interchangeable in the `io` APIs.

At the top of the I/O hierarchy is the abstract base class `I0Base`. It defines the basic interface to a stream. Note, however, that there is no separation between reading and writing to streams; implementations are allowed to raise an `IOError` if they do not support a given operation.

Extending `I0Base` is `RawI0Base` which deals simply with the reading and writing of raw bytes to a stream. `FileIO` subclasses `RawI0Base` to provide an interface to files in the machine’s file system.

`BufferedI0Base` deals with buffering on a raw byte stream (`RawI0Base`). Its subclasses, `BufferedReader`, `BufferedWriter`, and `BufferedReaderRWPair` buffer streams that are readable, writable, and both readable and writable. `BufferedReaderRandom` provides a buffered interface to random access streams. `BytesIO` is a simple stream of in-memory
bytes.

Another **IOBase** subclass, **TextIOBase**, deals with streams whose bytes represent text, and handles encoding and decoding from and to **unicode** strings. **TextIOWrapper**, which extends it, is a buffered text interface to a buffered raw stream (**BufferedI0Base**). Finally, **StringIO** is an in-memory stream for **unicode** text.

Argument names are not part of the specification, and only the arguments of **open()** are intended to be used as keyword arguments.
15.2.1. Module Interface

io. **DEFAULT_BUFFER_SIZE**

An int containing the default buffer size used by the module's buffered I/O classes. `open()` uses the file's blksize (as obtained by `os.stat()`) if possible.

io. **open**(*file*, *mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True*)

Open *file* and return a corresponding stream. If the file cannot be opened, an **IOError** is raised.

*file* is either a string giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless *closefd* is set to **False**.)

*mode* is an optional string that specifies the mode in which the file is opened. It defaults to 'r', which means open for reading in text mode. Other common values are 'w' for writing (truncating the file if it already exists), and 'a' for appending (which on some Unix systems, means that all writes append to the end of the file regardless of the current seek position). In text mode, if *encoding* is not specified the encoding used is platform dependent. (For reading and writing raw bytes use binary mode and leave *encoding* unspecified.) The available modes are:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r'</td>
<td>open for reading (default)</td>
</tr>
<tr>
<td>'w'</td>
<td>open for writing, truncating the file first</td>
</tr>
<tr>
<td>'a'</td>
<td>open for writing, appending to the end of the file if it exists</td>
</tr>
</tbody>
</table>
The default mode is 'rt' (open for reading text). For binary random access, the mode 'w+b' opens and truncates the file to 0 bytes, while 'r+b' opens the file without truncation.

Python distinguishes between files opened in binary and text modes, even when the underlying operating system doesn’t. Files opened in binary mode (including 'b' in the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when 't' is included in the mode argument), the contents of the file are returned as unicode strings, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size chunk buffer. When no buffering argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device’s “block size” and falling back on DEFAULT_BUFFER_SIZE. On many systems, the buffer will typically be 4096 or 8192 bytes long.
- “Interactive” text files (files for which isatty() returns True) use line buffering. Other text files use the policy described above for binary files.
**encoding** is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent (whatever `locale.getpreferredencoding()` returns), but any encoding supported by Python can be used. See the [codecs module](https://docs.python.org/3/library/codecs.html) for the list of supported encodings.

**errors** is an optional string that specifies how encoding and decoding errors are to be handled—this cannot be used in binary mode. Pass `strict` to raise a `ValueError` exception if there is an encoding error (the default of `None` has the same effect), or pass `ignore` to ignore errors. (Note that ignoring encoding errors can lead to data loss.) `replace` causes a replacement marker (such as `?`) to be inserted where there is malformed data. When writing, `xmlcharrefreplace` (replace with the appropriate XML character reference) or `backslashreplace` (replace with backslashed escape sequences) can be used. Any other error handling name that has been registered with `codecs.register_error()` is also valid.

**newline** controls how universal newlines works (it only applies to text mode). It can be `None`, `'`, `
`, ``, and `
`. It works as follows:

- On input, if `newline` is `None`, universal newlines mode is enabled. Lines in the input can end in `
`, ``, or `
`, and these are translated into `
` before being returned to the caller. If it is `'`, universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- On output, if `newline` is `None`, any `
` characters written are
translated to the system default line separator, \texttt{os.linesep}. If \texttt{newline} is '', no translation takes place. If \texttt{newline} is any of the other legal values, any \texttt{'\n'} characters written are translated to the given string.

If \texttt{closefd} is \texttt{False} and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given \texttt{closefd} has no effect and must be \texttt{True} (the default).

The type of file object returned by the \texttt{open()} function depends on the mode. When \texttt{open()} is used to open a file in a text mode (\texttt{'w'}, \texttt{'r'}, \texttt{'wt'}, \texttt{'rt'}, etc.), it returns a subclass of \texttt{TextIOWrapper} (specifically \texttt{TextIOBase}). When used to open a file in a binary mode with buffering, the returned class is a subclass of \texttt{BufferedIOBase}. The exact class varies: in read binary mode, it returns a \texttt{BufferedReader}; in write binary and append binary modes, it returns a \texttt{BufferedWriter}, and in read/write mode, it returns a \texttt{BufferedRandom}. When buffering is disabled, the raw stream, a subclass of \texttt{RawIOBase, FileIO}, is returned.

It is also possible to use an \texttt{unicode} or \texttt{bytes} string as a file for both reading and writing. For \texttt{unicode} strings \texttt{StringIO} can be used like a file opened in text mode, and for \texttt{bytes} a \texttt{BytesIO} can be used like a file opened in a binary mode.

\texttt{exception \texttt{io.BlockingIOError}}

Error raised when blocking would occur on a non-blocking stream. It inherits \texttt{IOError}.

In addition to those of \texttt{IOError}, \texttt{BlockingIOError} has one attribute:

\texttt{characters_written}
An integer containing the number of characters written to the stream before it blocked.

An exception inheriting IOError and ValueError that is raised when an unsupported operation is called on a stream.
class io.IOBase

The abstract base class for all I/O classes, acting on streams of bytes. There is no public constructor.

This class provides empty abstract implementations for many methods that derived classes can override selectively; the default implementations represent a file that cannot be read, written or seeked.

Even though IOBase does not declare read(), readinto(), or write() because their signatures will vary, implementations and clients should consider those methods part of the interface. Also, implementations may raise an IOError when operations they do not support are called.

The basic type used for binary data read from or written to a file is bytes (also known as str). Method arguments may also be bytearray or memoryview of arrays of bytes. In some cases, such as readinto(), a writable object such as bytearray is required. Text I/O classes work with unicode data.

*Changed in version 2.7:* Implementations should support memoryview arguments.

Note that calling any method (even inquiries) on a closed stream is undefined. Implementations may raise IOError in this case.

IOBase (and its subclasses) support the iterator protocol, meaning that an IOBase object can be iterated over yielding the lines in a stream. Lines are defined slightly differently depending on whether the stream is a binary stream (yielding bytes), or a
text stream (yielding `unicode` strings). See `readline()` below.

IOBase is also a context manager and therefore supports the `with` statement. In this example, `file` is closed after the `with` statement’s suite is finished—even if an exception occurs:

```python
with io.open('spam.txt', 'w') as file:
    file.write(u'Spam and eggs!')
```

`IOBase` provides these data attributes and methods:

- **close()**
  
  Flush and close this stream. This method has no effect if the file is already closed. Once the file is closed, any operation on the file (e.g. reading or writing) will raise a `ValueError`.

  As a convenience, it is allowed to call this method more than once; only the first call, however, will have an effect.

- **closed**
  
  True if the stream is closed.

- **fileno()**
  
  Return the underlying file descriptor (an integer) of the stream if it exists. An `IOError` is raised if the IO object does not use a file descriptor.

- **flush()**
  
  Flush the write buffers of the stream if applicable. This does nothing for read-only and non-blocking streams.

- **isatty()**
  
  Return `True` if the stream is interactive (i.e., connected to a terminal/tty device).

- **readable()**
Return `True` if the stream can be read from. If `False`, `read()` will raise `IOError`.

`readline(limit=-1)`
Read and return one line from the stream. If `limit` is specified, at most `limit` bytes will be read.

The line terminator is always `b'\n'` for binary files; for text files, the `newline` argument to `open()` can be used to select the line terminator(s) recognized.

`readlines(hint=-1)`
Read and return a list of lines from the stream. `hint` can be specified to control the number of lines read: no more lines will be read if the total size (in bytes/characters) of all lines so far exceeds `hint`.

Note that it's already possible to iterate on file objects using `for line in file: ...` without calling `file.readlines()`.

`seek(offset[, whence])`
Change the stream position to the given byte `offset`. `offset` is interpreted relative to the position indicated by `whence`. The default value for `whence` is `SEEK_SET`. Values for `whence` are:

- `SEEK_SET` or `0` – start of the stream (the default); `offset` should be zero or positive
- `SEEK_CUR` or `1` – current stream position; `offset` may be negative
- `SEEK_END` or `2` – end of the stream; `offset` is usually negative

Return the new absolute position.

*New in version 2.7:* The `SEEK_*` constants
seekable()
Return True if the stream supports random access. If False, seek(), tell() and truncate() will raise IOError.

tell()
Return the current stream position.

truncate(size=None)
Resize the stream to the given size in bytes (or the current position if size is not specified). The current stream position isn't changed. This resizing can extend or reduce the current file size. In case of extension, the contents of the new file area depend on the platform (on most systems, additional bytes are zero-filled, on Windows they're undetermined). The new file size is returned.

writable()
Return True if the stream supports writing. If False, write() and truncate() will raise IOError.

writelines(lines)
Write a list of lines to the stream. Line separators are not added, so it is usual for each of the lines provided to have a line separator at the end.

__del__()
Prepare for object destruction. I/OBase provides a default implementation of this method that calls the instance’s close() method.

class io.RawIOBase
Base class for raw binary I/O. It inherits I/OBase. There is no public constructor.
Raw binary I/O typically provides low-level access to an underlying OS device or API, and does not try to encapsulate it in high-level primitives (this is left to Buffered I/O and Text I/O, described later in this page).

In addition to the attributes and methods from `IOBase`, `RawIOBase` provides the following methods:

**read(n=-1)**

Read up to n bytes from the object and return them. As a convenience, if n is unspecified or -1, `readall()` is called. Otherwise, only one system call is ever made. Fewer than n bytes may be returned if the operating system call returns fewer than n bytes.

If 0 bytes are returned, and n was not 0, this indicates end of file. If the object is in non-blocking mode and no bytes are available, `None` is returned.

**readall()**

Read and return all the bytes from the stream until EOF, using multiple calls to the stream if necessary.

**readinto(b)**

Read up to len(b) bytes into b, and return the number of bytes read. The object b should be a pre-allocated, writable array of bytes, either `bytearray` or `memoryview`. If the object is in non-blocking mode and no bytes are available, `None` is returned.

**write(b)**

Write b to the underlying raw stream, and return the number of bytes written. The object b should be an array of bytes, either `bytes`, `bytearray`, or `memoryview`. The return value can be less than `len(b)`, depending on specifics of the underlying
raw stream, and especially if it is in non-blocking mode. None is returned if the raw stream is set not to block and no single byte could be readily written to it. The caller may release or mutate b after this method returns, so the implementation should only access b during the method call.

```python
class io.BufferedIOBase

Base class for binary streams that support some kind of buffering. It inherits IOBase. There is no public constructor.

The main difference with RawIOBase is that methods read(), readinto() and write() will try (respectively) to read as much input as requested or to consume all given output, at the expense of making perhaps more than one system call.

In addition, those methods can raise BlockingIOError if the underlying raw stream is in non-blocking mode and cannot take or give enough data; unlike their RawIOBase counterparts, they will never return None.

Besides, the read() method does not have a default implementation that defers to readinto().

A typical BufferedIOBase implementation should not inherit from a RawIOBase implementation, but wrap one, like BufferedWriter and BufferedReader do.

BufferedIOBase provides or overrides these methods and attribute in addition to those from IOBase:

raw

The underlying raw stream (a RawIOBase instance) that BufferedIOBase deals with. This is not part of the BufferedIOBase API and may not exist on some
implementations.

**detach()**
Separate the underlying raw stream from the buffer and return it.

After the raw stream has been detached, the buffer is in an unusable state.

Some buffers, like `BytesIO`, do not have the concept of a single raw stream to return from this method. They raise `UnsupportedOperation`.

*New in version 2.7.*

**read(n=-1)**
Read and return up to n bytes. If the argument is omitted, `None`, or negative, data is read and returned until EOF is reached. An empty bytes object is returned if the stream is already at EOF.

If the argument is positive, and the underlying raw stream is not interactive, multiple raw reads may be issued to satisfy the byte count (unless EOF is reached first). But for interactive raw streams, at most one raw read will be issued, and a short result does not imply that EOF is imminent.

A `BlockingIOError` is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

**read1(n=-1)**
Read and return up to n bytes, with at most one call to the underlying raw stream’s `read()` method. This can be useful if you are implementing your own buffering on top of a `BufferedIOBase` object.
**readinto(b)**

Read up to len(b) bytes into b, and return the number of bytes read. The object b should be a pre-allocated, writable array of bytes, either bytearray or memoryview.

Like `read()`, multiple reads may be issued to the underlying raw stream, unless the latter is ‘interactive’.

A *BlockingIOError* is raised if the underlying raw stream is in non-blocking-mode, and has no data available at the moment.

**write(b)**

Write b, and return the number of bytes written (always equal to len(b), since if the write fails an *IOError* will be raised). The object b should be an array of bytes, either bytes, bytearray, or memoryview. Depending on the actual implementation, these bytes may be readily written to the underlying stream, or held in a buffer for performance and latency reasons.

When in non-blocking mode, a *BlockingIOError* is raised if the data needed to be written to the raw stream but it couldn’t accept all the data without blocking.

The caller may release or mutate b after this method returns, so the implementation should only access b during the method call.
15.2.3. Raw File I/O

```python
class io.FileIO(name, mode='r', closefd=True)
```

FileIO represents an OS-level file containing bytes data. It implements the RawIOBase interface (and therefore the IOBase interface, too).

The name can be one of two things:

- a string representing the path to the file which will be opened;
- an integer representing the number of an existing OS-level file descriptor to which the resulting FileIO object will give access.

The mode can be 'r', 'w' or 'a' for reading (default), writing, or appending. The file will be created if it doesn’t exist when opened for writing or appending; it will be truncated when opened for writing. Add a '+' to the mode to allow simultaneous reading and writing.

The read() (when called with a positive argument), readinto() and write() methods on this class will only make one system call.

In addition to the attributes and methods from IOBase and RawIOBase, FileIO provides the following data attributes and methods:

**mode**

The mode as given in the constructor.

**name**

The file name. This is the file descriptor of the file when no

name is given in the constructor.
15.2.4. Buffered Streams

Buffered I/O streams provide a higher-level interface to an I/O device than raw I/O does.

```python
class io.BytesIO([initial_bytes])
```

A stream implementation using an in-memory bytes buffer. It inherits `BufferedIOBase`.

The optional argument `initial_bytes` is a `bytes` object that contains initial data.

`BytesIO` provides or overrides these methods in addition to those from `BufferedIOBase` and `IOBase`:

- `getvalue()`
  Return `bytes` containing the entire contents of the buffer.

- `read1()`
  In `BytesIO`, this is the same as `read()`.

```python
class io.BufferedReader(raw, buffer_size=DEFAULT_BUFFER_SIZE)
```

A buffer providing higher-level access to a readable, sequential `RawIOBase` object. It inherits `BufferedIOBase`. When reading data from this object, a larger amount of data may be requested from the underlying raw stream, and kept in an internal buffer. The buffered data can then be returned directly on subsequent reads.

The constructor creates a `BufferedReader` for the given readable `raw` stream and `buffer_size`. If `buffer_size` is omitted, `DEFAULT_BUFFER_SIZE` is used.
BufferedReader provides or overrides these methods in addition to those from BufferedReader and IOBase:

peek([n])
Return bytes from the stream without advancing the position. At most one single read on the raw stream is done to satisfy the call. The number of bytes returned may be less or more than requested.

read([n])
Read and return n bytes, or if n is not given or negative, until EOF or if the read call would block in non-blocking mode.

read1(n)
Read and return up to n bytes with only one call on the raw stream. If at least one byte is buffered, only buffered bytes are returned. Otherwise, one raw stream read call is made.

class io.BufferedReader(raw, buffer_size=DEFAULT_BUFFER_SIZE)
A buffer providing higher-level access to a writeable, sequential RawIOBase object. It inherits BufferedReader. When writing to this object, data is normally held into an internal buffer. The buffer will be written out to the underlying RawIOBase object under various conditions, including:

- when the buffer gets too small for all pending data;
- when flush() is called;
- when a seek() is requested (for BufferedRandom objects);
- when the BufferedWriter object is closed or destroyed.

The constructor creates a BufferedWriter for the given writeable raw stream. If the buffer_size is not given, it defaults to DEFAULT_BUFFER_SIZE.
A third argument, `max_buffer_size`, is supported, but unused and deprecated.

`BufferedWriter` provides or overrides these methods in addition to those from `BufferedIOBase` and `IOBase`:

- **flush()**
  - Force bytes held in the buffer into the raw stream. A `BlockingIOError` should be raised if the raw stream blocks.

- **write(b)**
  - Write `b`, and return the number of bytes written. The object `b` should be an array of bytes, either `bytes`, `bytearray`, or `memoryview`. When in non-blocking mode, a `BlockingIOError` is raised if the buffer needs to be written out but the raw stream blocks.

```python
class io.BufferedRandom(raw, 
buffer_size=DEFAULT_BUFFER_SIZE)
```

- A buffered interface to random access streams. It inherits `BufferedReader` and `BufferedWriter`, and further supports `seek()` and `tell()` functionality.

  The constructor creates a reader and writer for a seekable raw stream, given in the first argument. If the `buffer_size` is omitted it defaults to `DEFAULT_BUFFER_SIZE`.

  A third argument, `max_buffer_size`, is supported, but unused and deprecated.

  `BufferedRandom` is capable of anything `BufferedReader` or `BufferedWriter` can do.

```python
class io.BufferedRWPair(reader, writer, 
buffer_size=DEFAULT_BUFFER_SIZE)
```
A buffered I/O object combining two unidirectional `RawIOBase` objects – one readable, the other writeable – into a single bidirectional endpoint. It inherits `BufferedIOBase`.

*reader* and *writer* are `RawIOBase` objects that are readable and writeable respectively. If the `buffer_size` is omitted it defaults to `DEFAULT_BUFFER_SIZE`.

A fourth argument, `max_buffer_size`, is supported, but unused and deprecated.

`BufferedRWPair` implements all of `BufferedIOBase`'s methods except for `detach()`, which raises `UnsupportedOperation`.

**Warning:** `BufferedRWPair` does not attempt to synchronize accesses to its underlying raw streams. You should not pass it the same object as reader and writer; use `BufferedRandom` instead.
15.2.5. Text I/O

class io.TextIOBase
Base class for text streams. This class provides a unicode character and line based interface to stream I/O. There is no readinto() method because Python’s unicode strings are immutable. It inherits IOBase. There is no public constructor.

TextIOBase provides or overrides these data attributes and methods in addition to those from IOBase:

encoding
The name of the encoding used to decode the stream’s bytes into strings, and to encode strings into bytes.

errors
The error setting of the decoder or encoder.

newlines
A string, a tuple of strings, or None, indicating the newlines translated so far. Depending on the implementation and the initial constructor flags, this may not be available.

buffer
The underlying binary buffer (a BufferedIOBase instance) that TextIOBase deals with. This is not part of the TextIOBase API and may not exist on some implementations.

detach()
Separate the underlying binary buffer from the TextIOBase and return it.

After the underlying buffer has been detached, the TextIOBase is in an unusable state.
Some `TextIOWBase` implementations, like `StringIO`, may not have the concept of an underlying buffer and calling this method will raise `UnsupportedOperationException`.

`New in version 2.7.`

`read(n)`
Read and return at most `n` characters from the stream as a single `unicode`. If `n` is negative or `None`, reads until EOF.

`readline(limit=-1)`
Read until newline or EOF and return a single `unicode`. If the stream is already at EOF, an empty string is returned.

If `limit` is specified, at most `limit` characters will be read.

`seek(offset[, whence])`
Change the stream position to the given `offset`. Behaviour depends on the `whence` parameter. The default value for `whence` is `SEEK_SET`.

- `SEEK_SET` or `0`: seek from the start of the stream (the default); `offset` must either be a number returned by `TextIOWBase.tell()`, or zero. Any other `offset` value produces undefined behaviour.
- `SEEK_CUR` or `1`: “seek” to the current position; `offset` must be zero, which is a no-operation (all other values are unsupported).
- `SEEK_END` or `2`: seek to the end of the stream; `offset` must be zero (all other values are unsupported).

Return the new absolute position as an opaque number.

`New in version 2.7`: The `SEEK_*` constants.
tell()
Return the current stream position as an opaque number. The number does not usually represent a number of bytes in the underlying binary storage.

write(s)
Write the unicode string s to the stream and return the number of characters written.

class io.TextIOWrapper(buffer, encoding=None, errors=None, newline=None, line_buffering=False)
A buffered text stream over a BufferedIOBase binary stream. It inherits TextIOBase.

encoding gives the name of the encoding that the stream will be decoded or encoded with. It defaults to locale.getpreferredencoding().

errors is an optional string that specifies how encoding and decoding errors are to be handled. Pass 'strict' to raise a ValueError exception if there is an encoding error (the default of None has the same effect), or pass 'ignore' to ignore errors. (Note that ignoring encoding errors can lead to data loss.) 'replace' causes a replacement marker (such as '?') to be inserted where there is malformed data. When writing, 'xmlcharrefreplace' (replace with the appropriate XML character reference) or 'backslashreplace' (replace with backslashed escape sequences) can be used. Any other error handling name that has been registered with codecs.register_error() is also valid.

newline controls how line endings are handled. It can be None, '', '\n', '\r', and '\r\n'. It works as follows:
• On input, if `newline` is `None`, universal newlines mode is enabled. Lines in the input can end in `'\n'`, `'\r'`, or `'\r\n'`, and these are translated into `'\n'` before being returned to the caller. If it is `'\n'`, universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.

• On output, if `newline` is `None`, any `'\n'` characters written are translated to the system default line separator, `os.linesep`. If `newline` is `'\n'`, no translation takes place. If `newline` is any of the other legal values, any `'\n'` characters written are translated to the given string.

If `line_buffering` is `True`, `flush()` is implied when a call to `write` contains a newline character.

`TextIOWrapper` provides one attribute in addition to those of `TextIOBase` and its parents:

`line_buffering`
Whether line buffering is enabled.

```python
class io.StringIO(initial_value=u'', newline=u'\n')
```
An in-memory stream for unicode text. It inherits `TextIOWrapper`.

The initial value of the buffer can be set by providing `initial_value`. If newline translation is enabled, newlines will be encoded as if by `write()`. The stream is positioned at the start of the buffer.

The `newline` argument works like that of `TextIOWrapper`. The default is to consider only `\n` characters as ends of lines and to do no newline translation. If `newline` is set to `None`, newlines are written as `\n` on all platforms, but universal newline decoding is
still performed when reading.

**StringIO** provides this method in addition to those from **TextIOWrapper** and its parents:

**getvalue()**

Return a **unicode** containing the entire contents of the buffer at any time before the **StringIO** object’s **close()** method is called. Newlines are decoded as if by **read()**, although the stream position is not changed.

Example usage:

```python
import io
output = io.StringIO()
output.write(u'First line.
')
output.write(u'Second line.
')

# Retrieve file contents -- this will be
# u'First line.
Second line.
' contents = output.getvalue()

# Close object and discard memory buffer --
# ..getvalue() will now raise an exception.
output.close()
```

class **io.IncrementalNewlineDecoder**

A helper codec that decodes newlines for **universal newlines** mode. It inherits **codecs.IncrementalDecoder**.
15.2.6. Advanced topics

Here we will discuss several advanced topics pertaining to the concrete I/O implementations described above.

15.2.6.1. Performance

15.2.6.1.1. Binary I/O

By reading and writing only large chunks of data even when the user asks for a single byte, buffered I/O is designed to hide any inefficiency in calling and executing the operating system’s unbuffered I/O routines. The gain will vary very much depending on the OS and the kind of I/O which is performed (for example, on some contemporary OSes such as Linux, unbuffered disk I/O can be as fast as buffered I/O). The bottom line, however, is that buffered I/O will offer you predictable performance regardless of the platform and the backing device. Therefore, it is most always preferable to use buffered I/O rather than unbuffered I/O.

15.2.6.1.2. Text I/O

Text I/O over a binary storage (such as a file) is significantly slower than binary I/O over the same storage, because it implies conversions from unicode to binary data using a character codec. This can become noticeable if you handle huge amounts of text data (for example very large log files). Also, TextIOWrapper.tell() and TextIOWrapper.seek() are both quite slow due to the reconstruction algorithm used.

StringIO, however, is a native in-memory unicode container and will exhibit similar speed to BytesIO.
15.2.6.2. Multi-threading

FileIO objects are thread-safe to the extent that the operating system calls (such as `read(2)` under Unix) they are wrapping are thread-safe too.

Binary buffered objects (instances of `BufferedReader`, `BufferedWriter`, `BufferedRandom` and `BufferedRWPair`) protect their internal structures using a lock; it is therefore safe to call them from multiple threads at once.

TextIOWrapper objects are not thread-safe.

15.2.6.3. Reentrancy

Binary buffered objects (instances of `BufferedReader`, `BufferedWriter`, `BufferedRandom` and `BufferedRWPair`) are not reentrant. While reentrant calls will not happen in normal situations, they can arise if you are doing I/O in a `signal` handler. If it is attempted to enter a buffered object again while already being accessed `from the same thread`, then a `RuntimeError` is raised.

The above implicitly extends to text files, since the `open()` function will wrap a buffered object inside a `TextIOWrapper`. This includes standard streams and therefore affects the built-in function `print()` as well.
15.3. time — Time access and conversions

This module provides various time-related functions. For related functionality, see also the datetime and calendar modules.

Although this module is always available, not all functions are available on all platforms. Most of the functions defined in this module call platform C library functions with the same name. It may sometimes be helpful to consult the platform documentation, because the semantics of these functions varies among platforms.

An explanation of some terminology and conventions is in order.

- **The epoch** is the point where the time starts. On January 1st of that year, at 0 hours, the “time since the epoch” is zero. For Unix, the epoch is 1970. To find out what the epoch is, look at `gmtime(0)`.

- The functions in this module do not handle dates and times before the epoch or far in the future. The cut-off point in the future is determined by the C library; for Unix, it is typically in 2038.

- **Year 2000 (Y2K) issues**: Python depends on the platform’s C library, which generally doesn’t have year 2000 issues, since all dates and times are represented internally as seconds since the epoch. Functions accepting a `struct_time` (see below) generally require a 4-digit year. For backward compatibility, 2-digit years are supported if the module variable `accept2dyear` is a non-zero integer; this variable is initialized to 1 unless the environment variable `PYTHONY2K` is set to a non-empty string, in which case it is initialized to 0. Thus, you can set `PYTHONY2K` to a non-empty
string in the environment to require 4-digit years for all year input. When 2-digit years are accepted, they are converted according to the POSIX or X/Open standard: values 69-99 are mapped to 1969-1999, and values 0–68 are mapped to 2000–2068. Values 100–1899 are always illegal.

- UTC is Coordinated Universal Time (formerly known as Greenwich Mean Time, or GMT). The acronym UTC is not a mistake but a compromise between English and French.

- DST is Daylight Saving Time, an adjustment of the timezone by (usually) one hour during part of the year. DST rules are magic (determined by local law) and can change from year to year. The C library has a table containing the local rules (often it is read from a system file for flexibility) and is the only source of True Wisdom in this respect.

- The precision of the various real-time functions may be less than suggested by the units in which their value or argument is expressed. E.g. on most Unix systems, the clock “ticks” only 50 or 100 times a second.

- On the other hand, the precision of \texttt{time()} and \texttt{sleep()} is better than their Unix equivalents: times are expressed as floating point numbers, \texttt{time()} returns the most accurate time available (using Unix \texttt{gettimeofday()} where available), and \texttt{sleep()} will accept a time with a nonzero fraction (Unix \texttt{select()} is used to implement this, where available).

- The time value as returned by \texttt{gmtime()}, \texttt{localtime()}, and \texttt{strftime()}, and accepted by \texttt{asctime()}, \texttt{mktime()} and \texttt{strftime()}, may be considered as a sequence of 9 integers. The return values of \texttt{gmtime()}, \texttt{localtime()}, and \texttt{strftime()} also offer attribute names for individual fields.
See **struct_time** for a description of these objects.

*Changed in version 2.2:* The time value sequence was changed from a tuple to a **struct_time**, with the addition of attribute names for the fields.

- Use the following functions to convert between time representations:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>seconds since the epoch</td>
<td><strong>struct_time</strong> in UTC</td>
<td>gmtime()</td>
</tr>
<tr>
<td>seconds since the epoch</td>
<td><strong>struct_time</strong> in local time</td>
<td>localtime()</td>
</tr>
<tr>
<td><strong>struct_time</strong> in UTC</td>
<td>seconds since the epoch</td>
<td>calendar.timegm()</td>
</tr>
<tr>
<td><strong>struct_time</strong> in local time</td>
<td>seconds since the epoch</td>
<td>mktime()</td>
</tr>
</tbody>
</table>

The module defines the following functions and data items:

**time.** accept2dyear

Boolean value indicating whether two-digit year values will be accepted. This is true by default, but will be set to false if the environment variable **PYTHON2K** has been set to a non-empty string. It may also be modified at run time.

**time.** altzone

The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if **daylight** is nonzero.

**time.** asctime([t])

Convert a tuple or **struct_time** representing a time as returned
by `gmtime()` or `localtime()` to a 24-character string of the following form: 'Sun Jun 20 23:21:05 1993'. If `t` is not provided, the current time as returned by `localtime()` is used. Locale information is not used by `asctime()`.

**Note:** Unlike the C function of the same name, there is no trailing newline.

*Changed in version 2.1:* Allowed `t` to be omitted.

**time.clock()**

On Unix, return the current processor time as a floating point number expressed in seconds. The precision, and in fact the very definition of the meaning of “processor time”, depends on that of the C function of the same name, but in any case, this is the function to use for benchmarking Python or timing algorithms.

On Windows, this function returns wall-clock seconds elapsed since the first call to this function, as a floating point number, based on the Win32 function `QueryPerformanceCounter()`. The resolution is typically better than one microsecond.

**time.ctime([secs])**

Convert a time expressed in seconds since the epoch to a string representing local time. If `secs` is not provided or `None`, the current time as returned by `time()` is used. `ctime(secs)` is equivalent to `asctime(localtime(secs))`. Locale information is not used by `ctime()`.

*Changed in version 2.1:* Allowed `secs` to be omitted.

*Changed in version 2.4:* If `secs` is `None`, the current time is used.

**time.daylight**
Nonzero if a DST timezone is defined.

\texttt{time.gmtime([secs])}

Convert a time expressed in seconds since the epoch to a \texttt{struct_time} in UTC in which the dst flag is always zero. If \texttt{secs} is not provided or \texttt{None}, the current time as returned by \texttt{time()} is used. Fractions of a second are ignored. See above for a description of the \texttt{struct_time} object. See \texttt{calendar.timegm()} for the inverse of this function.

\textit{Changed in version 2.1:} Allowed \texttt{secs} to be omitted.

\textit{Changed in version 2.4:} If \texttt{secs} is \texttt{None}, the current time is used.

\texttt{time.localtime([secs])}

Like \texttt{gmtime()} but converts to local time. If \texttt{secs} is not provided or \texttt{None}, the current time as returned by \texttt{time()} is used. The dst flag is set to 1 when DST applies to the given time.

\textit{Changed in version 2.1:} Allowed \texttt{secs} to be omitted.

\textit{Changed in version 2.4:} If \texttt{secs} is \texttt{None}, the current time is used.

\texttt{time.mktime(t)}

This is the inverse function of \texttt{localtime()}. Its argument is the \texttt{struct_time} or full 9-tuple (since the dst flag is needed; use -1 as the dst flag if it is unknown) which expresses the time in local time, not UTC. It returns a floating point number, for compatibility with \texttt{time()}. If the input value cannot be represented as a valid time, either \texttt{OverflowError} or \texttt{ValueError} will be raised (which depends on whether the invalid value is caught by Python or the underlying C libraries). The earliest date for which it can generate a time is platform-dependent.
time.sleep(secs)
Suspend execution of the current thread for the given number of
seconds. The argument may be a floating point number to
indicate a more precise sleep time. The actual suspension time
may be less than that requested because any caught signal will
terminate the sleep() following execution of that signal's catching
routine. Also, the suspension time may be longer than requested
by an arbitrary amount because of the scheduling of other activity
in the system.

time.strftime(format[, t])
Convert a tuple or struct_time representing a time as returned
by gmtime() or localtime() to a string as specified by the format
argument. If t is not provided, the current time as returned by
localtime() is used. format must be a string. ValueError is
raised if any field in t is outside of the allowed range. strftime()
returns a locale depedent byte string; the result may be
converted to unicode by doing
strftime(<myformat>).decode(locale.getlocale()[1]).

*Changed in version 2.1:* Allowed t to be omitted.

*Changed in version 2.4:* ValueError raised if a field in t is out of
range.

*Changed in version 2.5:* 0 is now a legal argument for any
position in the time tuple; if it is normally illegal the value is forced
to a correct one.

The following directives can be embedded in the format string.
They are shown without the optional field width and precision
specification, and are replaced by the indicated characters in the
strftime() result:
<table>
<thead>
<tr>
<th>Directive</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Locale's abbreviated weekday name.</td>
<td></td>
</tr>
<tr>
<td>%A</td>
<td>Locale's full weekday name.</td>
<td></td>
</tr>
<tr>
<td>%b</td>
<td>Locale's abbreviated month name.</td>
<td></td>
</tr>
<tr>
<td>%B</td>
<td>Locale's full month name.</td>
<td></td>
</tr>
<tr>
<td>%c</td>
<td>Locale's appropriate date and time representation.</td>
<td></td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month as a decimal number [01,31].</td>
<td></td>
</tr>
<tr>
<td>%H</td>
<td>Hour (24-hour clock) as a decimal number [00,23].</td>
<td></td>
</tr>
<tr>
<td>%I</td>
<td>Hour (12-hour clock) as a decimal number [01,12].</td>
<td></td>
</tr>
<tr>
<td>%j</td>
<td>Day of the year as a decimal number [001,366].</td>
<td></td>
</tr>
<tr>
<td>%m</td>
<td>Month as a decimal number [01,12].</td>
<td></td>
</tr>
<tr>
<td>%M</td>
<td>Minute as a decimal number [00,59].</td>
<td></td>
</tr>
<tr>
<td>%p</td>
<td>Locale's equivalent of either AM or PM.</td>
<td>(1)</td>
</tr>
<tr>
<td>%S</td>
<td>Second as a decimal number [00,61].</td>
<td>(2)</td>
</tr>
<tr>
<td>%U</td>
<td>Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.</td>
<td>(3)</td>
</tr>
<tr>
<td>%w</td>
<td>Weekday as a decimal number [0(Sunday),6].</td>
<td></td>
</tr>
<tr>
<td>%W</td>
<td>Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.</td>
<td>(3)</td>
</tr>
<tr>
<td>%x</td>
<td>Locale's appropriate date representation.</td>
<td></td>
</tr>
<tr>
<td>%X</td>
<td>Locale's appropriate time representation.</td>
<td></td>
</tr>
<tr>
<td>%y</td>
<td>Year without century as a decimal number [00,99].</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>%Y</td>
<td>Year with century as a decimal number.</td>
<td></td>
</tr>
<tr>
<td>%Z</td>
<td>Time zone name (no characters if no time zone exists).</td>
<td></td>
</tr>
<tr>
<td>%%</td>
<td>A literal '%' character.</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. When used with the `strftime()` function, the `%p` directive only affects the output hour field if the `%I` directive is used to parse the hour.
2. The range really is 0 to 61; this accounts for leap seconds and the (very rare) double leap seconds.
3. When used with the `strftime()` function, `%U` and `%W` are only used in calculations when the day of the week and the year are specified.

Here is an example, a format for dates compatible with that specified in the RFC 2822 Internet email standard. [1]

```python
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

Additional directives may be supported on certain platforms, but only the ones listed here have a meaning standardized by ANSI C. To see the full set of format codes supported on your platform, consult the `strftime(3)` documentation.

On some platforms, an optional field width and precision specification can immediately follow the initial '%' of a directive in the following order; this is also not portable. The field width is normally 2 except for %j where it is 3.

```python
time.strptime(string[, format])
```
Parse a string representing a time according to a format. The return value is a \texttt{struct\_time} as returned by \texttt{gmtime()} or \texttt{localtime()}.

The \texttt{format} parameter uses the same directives as those used by \texttt{strftime()}; it defaults to "\%a \%b \%d \%H:\%M:\%S \%Y" which matches the formatting returned by \texttt{ctime()}. If \texttt{string} cannot be parsed according to \texttt{format}, or if it has excess data after parsing, \texttt{ValueError} is raised. The default values used to fill in any missing data when more accurate values cannot be inferred are \((1900, 1, 1, 0, 0, 0, 0, 1, -1)\).

For example:

```python
>>> import time
>>> time.strptime("30 Nov 00", "\%d \%b \%y")
```

```
\texttt{time.struct\_time(tm\_year=2000, tm\_mon=11, tm\_mday=30, tm\_hour=0, tm\_min=0, tm\_sec=0, tm\_wday=3, tm\_yday=335, tm\_isdst=-1)}
```

Support for the \texttt{%Z} directive is based on the values contained in \texttt{tzname} and whether \texttt{daylight} is true. Because of this, it is platform-specific except for recognizing UTC and GMT which are always known (and are considered to be non-daylight savings timezones).

Only the directives specified in the documentation are supported. Because \texttt{strftime()} is implemented per platform it can sometimes offer more directives than those listed. But \texttt{strptime()} is independent of any platform and thus does not necessarily support all directives available that are not documented as supported.

\texttt{class time.struct\_time}

The type of the time value sequence returned by \texttt{gmtime()},

\texttt{struct\_time}


`localtime()`, and `strptime()`. It is an object with a named tuple interface: values can be accessed by index and by attribute name. The following values are present:

<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>tm_year</code></td>
<td>(for example, 1993)</td>
</tr>
<tr>
<td>1</td>
<td><code>tm_mon</code></td>
<td>range [1, 12]</td>
</tr>
<tr>
<td>2</td>
<td><code>tm_mday</code></td>
<td>range [1, 31]</td>
</tr>
<tr>
<td>3</td>
<td><code>tm_hour</code></td>
<td>range [0, 23]</td>
</tr>
<tr>
<td>4</td>
<td><code>tm_min</code></td>
<td>range [0, 59]</td>
</tr>
<tr>
<td>5</td>
<td><code>tm_sec</code></td>
<td>range [0, 61]: see (2) in <code>strftime()</code> description</td>
</tr>
<tr>
<td>6</td>
<td><code>tm_wday</code></td>
<td>range [0, 6], Monday is 0</td>
</tr>
<tr>
<td>7</td>
<td><code>tm_yday</code></td>
<td>range [1, 366]</td>
</tr>
<tr>
<td>8</td>
<td><code>tm_isdst</code></td>
<td>0, 1 or -1; see below</td>
</tr>
</tbody>
</table>

*New in version 2.2.*

Note that unlike the C structure, the month value is a range of [1, 12], not [0, 11]. A year value will be handled as described under Year 2000 (Y2K) issues above.

In calls to `mktime()`, `tm_isdst` may be set to 1 when daylight savings time is in effect, and 0 when it is not. A value of -1 indicates that this is not known, and will usually result in the correct state being filled in.

When a tuple with an incorrect length is passed to a function expecting a `struct_time`, or having elements of the wrong type, a `TypeError` is raised.

`time.time()`

Return the time in seconds since the epoch as a floating point number. Note that even though the time is always returned as a floating point number, not all systems provide time with a better
precision than 1 second. While this function normally returns non-decreasing values, it can return a lower value than a previous call if the system clock has been set back between the two calls.

**time.timezone**

The offset of the local (non-DST) timezone, in seconds west of UTC (negative in most of Western Europe, positive in the US, zero in the UK).

**time.tzname**

A tuple of two strings: the first is the name of the local non-DST timezone, the second is the name of the local DST timezone. If no DST timezone is defined, the second string should not be used.

**time.tzset()**

Resets the time conversion rules used by the library routines. The environment variable `TZ` specifies how this is done.

*New in version 2.3.*

Availability: Unix.

**Note:** Although in many cases, changing the `TZ` environment variable may affect the output of functions like `localtime()` without calling `tzset()`, this behavior should not be relied on. The `TZ` environment variable should contain no whitespace.

The standard format of the `TZ` environment variable is (whitespace added for clarity):

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

Where the components are:
std and dst

Three or more alphanumerics giving the timezone abbreviations. These will be propagated into time.tzname

offset

The offset has the form: ± hh[:mm[:ss]]. This indicates the value added the local time to arrive at UTC. If preceded by a ‘-‘, the timezone is east of the Prime Meridian; otherwise, it is west. If no offset follows dst, summer time is assumed to be one hour ahead of standard time.

start[/time], end[/time]

Indicates when to change to and back from DST. The format of the start and end dates are one of the following:

Jn

The Julian day n (1 <= n <= 365). Leap days are not counted, so in all years February 28 is day 59 and March 1 is day 60.

n

The zero-based Julian day (0 <= n <= 365). Leap days are counted, and it is possible to refer to February 29.

Mm.n.d

The d'th day (0 <= d <= 6) or week n of month m of the year (1 <= n <= 5, 1 <= m <= 12, where week 5 means “the last d day in month m” which may occur in either the fourth or the fifth week). Week 1 is the first week in which the d'th day occurs. Day zero is Sunday.

time has the same format as offset except that no leading sign (-‘ or ‘+) is allowed. The default, if time is not given, is 02:00:00.

>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
On many Unix systems (including *BSD, Linux, Solaris, and Darwin), it is more convenient to use the system's zoneinfo (tzfile(5)) database to specify the timezone rules. To do this, set the `TZ` environment variable to the path of the required timezone datafile, relative to the root of the systems 'zoneinfo' timezone database, usually located at `/usr/share/zoneinfo`. For example, 'US/Eastern', 'Australia/Melbourne', 'Egypt' or 'Europe/Amsterdam'.

---

**See also:**

**Module datetime**

More object-oriented interface to dates and times.

**Module locale**

Internationalization services. The locale setting affects the interpretation of many format specifiers in `strftime()` and `strptime()`.

**Module calendar**

General calendar-related functions. `timetgm()` is the inverse of `gmtime()` from this module.
The use of %z is now deprecated, but the %z escape that expands to the preferred hour/minute offset is not supported by all ANSI C libraries. Also, a strict reading of the original 1982 RFC 822 standard calls for a two-digit year (%y rather than %Y), but practice moved to 4-digit years long before the year 2000. After that, RFC 822 became obsolete and the 4-digit year has been first recommended by RFC 1123 and then mandated by RFC 2822.

[1]
15.4. argparse — Parser for command-line options, arguments and sub-commands

New in version 2.7.

**Source code:** Lib/argparse.py

The `argparse` module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires, and `argparse` will figure out how to parse those out of `sys.argv`. The `argparse` module also automatically generates help and usage messages and issues errors when users give the program invalid arguments.

**Tutorial**

This page contains the API reference information. For a more gentle introduction to Python command-line parsing, have a look at the [argparse tutorial](https://docs.python.org/3/library/argparse.html).
15.4.1. Example

The following code is a Python program that takes a list of integers and produces either the sum or the max:

```python
import argparse

parser = argparse.ArgumentParser(description='Process some integers.
parser.add_argument('integers', metavar='N', type=int, nargs='+', help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', const=sum, default=max, help='sum the integers (default: find the max)

args = parser.parse_args()
print(args.accumulate(args.integers))
```

Assuming the Python code above is saved into a file called `prog.py`, it can be run at the command line and provides useful help messages:

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
  N                  an integer for the accumulator

optional arguments:
  -h, --help         show this help message and exit
  --sum              sum the integers (default: find the max)
```

When run with the appropriate arguments, it prints either the sum or the max of the command-line integers:

```
$ python prog.py 1 2 3 4
4
```
If invalid arguments are passed in, it will issue an error:

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

The following sections walk you through this example.

### 15.4.1.1. Creating a parser

The first step in using the `argparse` is creating an `ArgumentParser` object:

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

The `ArgumentParser` object will hold all the information necessary to parse the command line into Python data types.

### 15.4.1.2. Adding arguments

Filling an `ArgumentParser` with information about program arguments is done by making calls to the `add_argument()` method. Generally, these calls tell the `ArgumentParser` how to take the strings on the command line and turn them into objects. This information is stored and used when `parse_args()` is called. For example:

```
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+', help='an integer for the accumulator')
>>> parser.add_argument('--sum', dest='accumulate', action='store_const', const=sum, default=max, help='sum the integers (default: find the max)')
```
Later, calling `parse_args()` will return an object with two attributes, `integers` and `accumulate`. The `integers` attribute will be a list of one or more ints, and the `accumulate` attribute will be either the `sum()` function, if `--sum` was specified at the command line, or the `max()` function if it was not.

### 15.4.1.3. Parsing arguments

`ArgumentParser` parses arguments through the `parse_args()` method. This will inspect the command line, convert each argument to the appropriate type and then invoke the appropriate action. In most cases, this means a simple `Namespace` object will be built up from attributes parsed out of the command line:

```python
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

In a script, `parse_args()` will typically be called with no arguments, and the `ArgumentParser` will automatically determine the command-line arguments from `sys.argv`.
class argparse.ArgumentParser(prog=None, usage=None,
description=None, epilog=None, parents=[],
formatter_class=argparse.HelpFormatter, prefix_chars='-',
fromfile_prefix_chars=None, argument_default=None,
conflict_handler='error', add_help=True)

Create a new `ArgumentParser` object. All parameters should be passed as keyword arguments. Each parameter has its own more detailed description below, but in short they are:

- **prog** - The name of the program (default: `sys.argv[0]`)
- **usage** - The string describing the program usage (default: generated from arguments added to parser)
- **description** - Text to display before the argument help (default: none)
- **epilog** - Text to display after the argument help (default: none)
- **parents** - A list of `ArgumentParser` objects whose arguments should also be included
- **formatter_class** - A class for customizing the help output
- **prefix_chars** - The set of characters that prefix optional arguments (default: `-`)
- **fromfile_prefix_chars** - The set of characters that prefix files from which additional arguments should be read (default: `None`)
- **argument_default** - The global default value for arguments (default: `None`)
- **conflict_handler** - The strategy for resolving conflicting optionals (usually unnecessary)
- **add_help** - Add a -h/--help option to the parser (default: `True`)

The following sections describe how each of these are used.
By default, `ArgumentParser` objects use `sys.argv[0]` to determine how to display the name of the program in help messages. This default is almost always desirable because it will make the help messages match how the program was invoked on the command line. For example, consider a file named `myprogram.py` with the following code:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

The help for this program will display `myprogram.py` as the program name (regardless of where the program was invoked from):

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help    show this help message and exit
  --foo FOO     foo help
$ cd ..
$ python subdir\myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help    show this help message and exit
  --foo FOO     foo help
```

To change this default behavior, another value can be supplied using the `prog=` argument to `ArgumentParser`:

```python
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

optional arguments:
  -h, --help    show this help message and exit
```
Note that the program name, whether determined from `sys.argv[0]` or from the `prog=` argument, is available to help messages using the `%(prog)s` format specifier.

```python
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

optional arguments:
  -h, --help    show this help message and exit
  --foo FOO    foo of the myprogram program
```

### 15.4.2.2. usage

By default, `ArgumentParser` calculates the usage message from the arguments it contains:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
  bar            bar help

optional arguments:
  -h, --help    show this help message and exit
  --foo [FOO]   foo help
```

The default message can be overridden with the `usage=` keyword argument:

```python
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
```
usage: PROG [options]

positional arguments:
   bar      bar help

optional arguments:
   -h, --help       show this help message and exit
   --foo [FOO]     foo help

The `%prog` format specifier is available to fill in the program name in your usage messages.

### 15.4.2.3. description

Most calls to the `ArgumentParser` constructor will use the `description=` keyword argument. This argument gives a brief description of what the program does and how it works. In help messages, the description is displayed between the command-line usage string and the help messages for the various arguments:

```python
>>> parser = argparse.ArgumentParser(description='A foo that bars')
```

```python
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
   -h, --help       show this help message and exit
```

By default, the description will be line-wrapped so that it fits within the given space. To change this behavior, see the `formatter_class` argument.

### 15.4.2.4. epilog

Some programs like to display additional description of the program after the description of the arguments. Such text can be specified
using the `epilog=` argument to `ArgumentParser`:

```python
g>> parser = argparse.ArgumentParser(
...    description='A foo that bars',
...    epilog="And that's how you'd foo a bar")
g>> parser.print_help()
usage: argparse.py [-h]
A foo that bars
optional arguments:
    -h, --help    show this help message and exit
And that's how you'd foo a bar
```

As with the `description` argument, the `epilog=` text is by default line-wrapped, but this behavior can be adjusted with the `formatter_class` argument to `ArgumentParser`.

### 15.4.2.5. parents

Sometimes, several parsers share a common set of arguments. Rather than repeating the definitions of these arguments, a single parser with all the shared arguments and passed to `parents=` argument to `ArgumentParser` can be used. The `parents=` argument takes a list of `ArgumentParser` objects, collects all the positional and optional actions from them, and adds these actions to the `ArgumentParser` object being constructed:

```python
g>> parent_parser = argparse.ArgumentParser(add_help=False)
g>> parent_parser.add_argument('--parent', type=int)
g>> foo_parser = argparse.ArgumentParser(parents=[parent_parser]
g>> foo_parser.add_argument('foo')
g>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

g>> bar_parser = argparse.ArgumentParser(parents=[parent_parser]
g>> bar_parser.add_argument('--bar')
```
Note that most parent parsers will specify `add_help=False`. Otherwise, the `ArgumentParser` will see two `--help` options (one in the parent and one in the child) and raise an error.

**Note:** You must fully initialize the parsers before passing them via `parents=`. If you change the parent parsers after the child parser, those changes will not be reflected in the child.

### 15.4.2.6. formatter_class

`ArgumentParser` objects allow the help formatting to be customized by specifying an alternate formatting class. Currently, there are three such classes:

```python
class argparse.RawDescriptionHelpFormatter
class argparse.RawTextHelpFormatter
class argparse.ArgumentDefaultsHelpFormatter
```

The first two allow more control over how textual descriptions are displayed, while the last automatically adds information about argument default values.

By default, `ArgumentParser` objects line-wrap the `description` and `epilog` texts in command-line help messages:

```python
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...     was indented weird
...     but that is okay''',
...     epilog='''
...     likewise for this epilog whose whitespace will
...     be cleaned up and whose words will be wrapped
```
... across a couple lines''')

>>> parser.print_help()
usage: PROG [-h]

description was indented weird but that is okay

optional arguments:
    -h, --help    show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and
will be wrapped across a couple lines

Passing `RawDescriptionHelpFormatter` as `formatter_class=indicates` that `description` and `epilog` are already correctly formatted and should not be line-wrapped:

>>> parser = argparse.ArgumentParser(
    ...    prog='PROG',
    ...    formatter_class=argparse.RawDescriptionHelpFormatter,
    ...    description=textwrap.dedent('''
        Please do not mess up this text!
        --------------------------------
        I have indented it
        exactly the way
        I want it
        '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
--------------------------------
I have indented it
   exactly the way
   I want it

optional arguments:
    -h, --help    show this help message and exit

`RawTextHelpFormatter` maintains whitespace for all sorts of help text, including argument descriptions.

The other formatter class available, `ArgumentDefaultsHelpFormatter`,
will add information about the default value of each of the arguments:

```python
>>> parser = argparse.ArgumentParser(
    ...    prog='PROG',
    ...    formatter_class=argparse.ArgumentDefaultsHelpFormatter)
```  
```python
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!
```  
```python
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!
```  
```python
>>> parser.print_help()
```

```python
usage: PROG [-h] [--foo FOO] [bar [bar ...]]
```

```python
positional arguments:
    bar    BAR! (default: [1, 2, 3])
```

```python
optional arguments:
    -h, --help    show this help message and exit
    --foo FOO    FOO! (default: 42)
```

15.4.2.7. prefix_chars

Most command-line options will use `-` as the prefix, e.g. `-f/--foo`. Parsers that need to support different or additional prefix characters, e.g. for options like `+f` or `/foo`, may specify them using the `prefix_chars=` argument to the ArgumentParser constructor:

```python
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
```  
```python
>>> parser.add_argument('+f')
```  
```python
>>> parser.add_argument('++bar')
```  
```python
>>> parser.parse_args('+f X ++bar Y'.split())
```

```python
Namespace(bar='Y', f='X')
```

The `prefix_chars=` argument defaults to `'-'`. Supplying a set of characters that does not include `-` will cause `-f/--foo` options to be disallowed.

15.4.2.8. fromfile_prefix_chars
Sometimes, for example when dealing with a particularly long argument lists, it may make sense to keep the list of arguments in a file rather than typing it out at the command line. If the `fromfile_prefix_chars=` argument is given to the `ArgumentParser` constructor, then arguments that start with any of the specified characters will be treated as files, and will be replaced by the arguments they contain. For example:

```python
>>> with open('args.txt', 'w') as fp:
    ...   fp.write('-f\nbar')
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

Arguments read from a file must by default be one per line (but see also `convert_arg_line_to_args()`) and are treated as if they were in the same place as the original file referencing argument on the command line. So in the example above, the expression `['-f', 'foo', '@args.txt']` is considered equivalent to the expression `['-f', 'foo', '-f', 'bar']`.

The `fromfile_prefix_chars=` argument defaults to `None`, meaning that arguments will never be treated as file references.

### 15.4.2.9. argument_default

Generally, argument defaults are specified either by passing a default to `add_argument()` or by calling the `set_defaults()` methods with a specific set of name-value pairs. Sometimes however, it may be useful to specify a single parser-wide default for arguments. This can be accomplished by passing the `argument_default=` keyword argument to `ArgumentParser`. For example, to globally suppress attribute creation on `parse_args()` calls, we supply
argument_default=SUPPRESS:

```python
g>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
g>>> parser.add_argument('--foo')
g>>> parser.add_argument('bar', nargs='?')
g>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
```  
```python
g>>> parser.parse_args([])
Namespace()
```  
15.4.2.10. conflict_handler

`ArgumentParser` objects do not allow two actions with the same option string. By default, `ArgumentParser` objects raise an exception if an attempt is made to create an argument with an option string that is already in use:

```python
g>>> parser = argparse.ArgumentParser(prog='PROG')
g>>> parser.add_argument('-f', '--foo', help='old foo help')
g>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
  ..
ArgumentError: argument --foo: conflicting option string(s): --
```  
Sometimes (e.g. when using `parents`) it may be useful to simply override any older arguments with the same option string. To get this behavior, the value `'resolve'` can be supplied to the `conflict_handler= argument of ArgumentParser`:

```python
g>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
g>>> parser.add_argument('-f', '--foo', help='old foo help')
g>>> parser.add_argument('--foo', help='new foo help')
g>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]
optional arguments:
  -h, --help    show this help message and exit
  -f FOO        old foo help
```
Note that `ArgumentParser` objects only remove an action if all of its option strings are overridden. So, in the example above, the old `-f/-foo` action is retained as the `-f` action, because only the `--foo` option string was overridden.

**15.4.2.11. add_help**

By default, `ArgumentParser` objects add an option which simply displays the parser's help message. For example, consider a file named `myprogram.py` containing the following code:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

If `-h` or `--help` is supplied at the command line, the `ArgumentParser` help will be printed:

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help    show this help message and exit
  --foo FOO    foo help
```

Occasionally, it may be useful to disable the addition of this help option. This can be achieved by passing `False` as the `add_help` argument to `ArgumentParser`:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [-foo FOO]
```
optional arguments:
   --foo FOO  foo help

The help option is typically `-h/--help`. The exception to this is if the
`prefix_chars=` is specified and does not include `-`, in which case `-h`
and `--help` are not valid options. In this case, the first character in
`prefix_chars` is used to prefix the help options:

```python
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/~
>>> parser.print_help()
usage: PROG [+h]

optional arguments:
   +h, ++help show this help message and exit
```
15.4.3. The add_argument() method

```
ArgumentParser.add_argument(name or flags..., action=, nargs=, const=, default=, type=, choices=, required=, help=, metavar=, dest=)
```

Define how a single command-line argument should be parsed. Each parameter has its own more detailed description below, but in short they are:

- **name or flags** - Either a name or a list of option strings, e.g. `foo` or `-f, --foo`.
- **action** - The basic type of action to be taken when this argument is encountered at the command line.
- **nargs** - The number of command-line arguments that should be consumed.
- **const** - A constant value required by some action and nargs selections.
- **default** - The value produced if the argument is absent from the command line.
- **type** - The type to which the command-line argument should be converted.
- **choices** - A container of the allowable values for the argument.
- **required** - Whether or not the command-line option may be omitted (optionals only).
- **help** - A brief description of what the argument does.
- **metavar** - A name for the argument in usage messages.
- **dest** - The name of the attribute to be added to the object returned by `parse_args()`.

The following sections describe how each of these are used.
15.4.3.1. name or flags

The `add_argument()` method must know whether an optional argument, like `-f` or `--foo`, or a positional argument, like a list of filenames, is expected. The first arguments passed to `add_argument()` must therefore be either a series of flags, or a simple argument name. For example, an optional argument could be created like:

```python
>>> parser.add_argument('-f', '--foo')
```

while a positional argument could be created like:

```python
>>> parser.add_argument('bar')
```

When `parse_args()` is called, optional arguments will be identified by the `-` prefix, and the remaining arguments will be assumed to be positional:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
```

```python
usage: PROG [-h] [-f FOO] bar
PROG: error: too few arguments
```

15.4.3.2. action

`ArgumentParser` objects associate command-line arguments with actions. These actions can do just about anything with the command-line arguments associated with them, though most actions simply add an attribute to the object returned by `parse_args()`. The
action keyword argument specifies how the command-line arguments should be handled. The supplied actions are:

- 'store' - This just stores the argument's value. This is the default action. For example:

  ```python
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo')
  >>> parser.parse_args(['--foo 1'].split())
  Namespace(foo='1')
  ```

- 'store_const' - This stores the value specified by the const keyword argument. The 'store_const' action is most commonly used with optional arguments that specify some sort of flag. For example:

  ```python
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', action='store_const', const=42)
  >>> parser.parse_args(['--foo'])
  Namespace(foo=42)
  ```

- 'store_true' and 'store_false' - These are special cases of 'store_const' using for storing the values True and False respectively. In addition, they create default values of False and True respectively. For example:

  ```python
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', action='store_true')
  >>> parser.add_argument('--bar', action='store_false')
  >>> parser.add_argument('--baz', action='store_false')
  >>> parser.parse_args(['--foo', '--bar'].split())
  Namespace(bar=False, baz=True, foo=True)
  ```

- 'append' - This stores a list, and appends each argument value to the list. This is useful to allow an option to be specified multiple times. Example usage:
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])

- **append_const** - This stores a list, and appends the value specified by the const keyword argument to the list. (Note that the const keyword argument defaults to None.) The append_const action is typically useful when multiple arguments need to store constants to the same list. For example:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const')
>>> parser.add_argument('--int', dest='types', action='append_const')
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<type 'str'>, <type 'int'>])
```

- **count** - This counts the number of times a keyword argument occurs. For example, this is useful for increasing verbosity levels:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count')
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

- **help** - This prints a complete help message for all the options in the current parser and then exits. By default a help action is automatically added to the parser. See ArgumentParser for details of how the output is created.

- **version** - This expects a version= keyword argument in the add_argument() call, and prints version information and exits when invoked:

```python
>>> import argparse
```
You may also specify an arbitrary action by passing an Action subclass or other object that implements the same interface. The recommended way to do this is to extend `Action`, overriding the `__call__` method and optionally the `__init__` method.

An example of a custom action:

```python
>>> class FooAction(argparse.Action):
    ...
    ...    def __init__(self, option_strings, dest, nargs=None, **kwargs):
    ...
    ...    if nargs is not None:
    ...        raise ValueError("nargs not allowed")
    ...
    ...    super(FooAction, self).__init__(option_strings, dest, nargs=None, **kwargs)
    ...
    ...    def __call__(self, parser, namespace, values, option_string=None):
    ...        print '%r %r %r' % (namespace, values, option_string)
    ...        setattr(namespace, self.dest, values)
    ...

>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

For more details, see `Action`.

### 15.4.3.3. nargs

ArgumentParser objects usually associate a single command-line argument with a single action to be taken. The `nargs` keyword argument associates a different number of command-line arguments with a single action. The supported values are:
- \( N \) (an integer). \( N \) arguments from the command line will be gathered together into a list. For example:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs=2)
>>> parser.add_argument('bar', nargs=1)
>>> parser.parse_args('c --foo a b'.split())
Namespace(bar=['c'], foo=['a', 'b'])
```

Note that \( \text{nargs=1} \) produces a list of one item. This is different from the default, in which the item is produced by itself.

- '??'. One argument will be consumed from the command line if possible, and produced as a single item. If no command-line argument is present, the value from default will be produced. Note that for optional arguments, there is an additional case - the option string is present but not followed by a command-line argument. In this case the value from const will be produced. Some examples to illustrate this:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', nargs='?', const='c', default=sys.stdin)
>>> parser.add_argument('outfile', nargs='?', const='d', default=sys.stdout)
>>> parser.parse_args(['input.txt', 'output.txt'])
```

One of the more common uses of \( \text{nargs='?'} \) is to allow optional input and output files:
### Namespace(infile=<open file 'input.txt', mode 'r' at 0x...>,
outfile=<open file 'output.txt', mode 'w' at 0x...>)

```python
>>> parser.parse_args([])
Namespace(infile=<open file '<stdin>', mode 'r' at 0x...>,
outfile=<open file '<stdout>', mode 'w' at 0x...>)
```

- **'*'**. All command-line arguments present are gathered into a list. Note that it generally doesn’t make much sense to have more than one positional argument with `nargs='*'`, but multiple optional arguments with `nargs='*'` is possible. For example:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args(['a b --foo x y --bar 1 2'].split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

- **'+'**. Just like `'*'`, all command-line args present are gathered into a list. Additionally, an error message will be generated if there wasn’t at least one command-line argument present. For example:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.add_argument('--foo')
>>> parser.add_argument('command')
>>> parser.add_argument('args', nargs=argparse.REMAINDER)
>>> print parser.parse_args('--foo B cmd --arg1 XX ZZ'.split())
```

- `argparse.REMAINDER`. All the remaining command-line arguments are gathered into a list. This is commonly useful for command line utilities that dispatch to other command line utilities:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo')
>>> parser.add_argument('command')
>>> parser.add_argument('args', nargs=argparse.REMAINDER)
>>> print parser.parse_args('--foo B cmd --arg1 XX ZZ'.split())
```
If the `nargs` keyword argument is not provided, the number of arguments consumed is determined by the `action`. Generally this means a single command-line argument will be consumed and a single item (not a list) will be produced.

### 15.4.3.4. `const`

The `const` argument of `add_argument()` is used to hold constant values that are not read from the command line but are required for the various `ArgumentParser` actions. The two most common uses of it are:

- When `add_argument()` is called with `action='store_const'` or `action='append_const'`. These actions add the `const` value to one of the attributes of the object returned by `parse_args()`. See the `action` description for examples.
- When `add_argument()` is called with option strings (like `-f` or `--foo`) and `nargs='?'`. This creates an optional argument that can be followed by zero or one command-line arguments. When parsing the command line, if the option string is encountered with no command-line argument following it, the value of `const` will be assumed instead. See the `nargs` description for examples.

With the `store_const` and `append_const` actions, the `const` keyword argument must be given. For other actions, it defaults to `None`.

### 15.4.3.5. `default`

All optional arguments and some positional arguments may be
omitted at the command line. The `default` keyword argument of `add_argument()`, whose value defaults to `None`, specifies what value should be used if the command-line argument is not present. For optional arguments, the `default` value is used when the option string was not present at the command line:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

If the `default` value is a string, the parser parses the value as if it were a command-line argument. In particular, the parser applies any `type` conversion argument, if provided, before setting the attribute on the `Namespace` return value. Otherwise, the parser uses the value as is:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

For positional arguments with `nargs` equal to `?` or `*`, the `default` value is used when no command-line argument was present:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

Providing `default=argparse.SUPPRESS` causes no attribute to be added if the command-line argument was not present.
By default, `ArgumentParser` objects read command-line arguments in as simple strings. However, quite often the command-line string should instead be interpreted as another type, like a `float` or `int`. The `type` keyword argument of `add_argument()` allows any necessary type-checking and type conversions to be performed. Common built-in types and functions can be used directly as the value of the `type` argument:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

See the section on the `default` keyword argument for information on when the `type` argument is applied to default arguments.

To ease the use of various types of files, the `argparse` module provides the factory `FileType` which takes the `mode=` and `bfsizem` arguments of the `file` object. For example, `FileType('w')` can be used to create a writable file:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar', type=argparse.FileType('w'))
>>> parser.parse_args(['out.txt'])
Namespace(bar=<open file 'out.txt', mode 'w' at 0x...>)
```
type= can take any callable that takes a single string argument and returns the converted value:

```python
>>> def perfect_square(string):
...    value = int(string)
...    sqrt = math.sqrt(value)
...    if sqrt != int(sqrt):
...        msg = "%r is not a perfect square" % string
...        raise argparse.ArgumentTypeError(msg)
...    return value
...```

The choices keyword argument may be more convenient for type checkers that simply check against a range of values:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=perfect_square)
>>> parser.parse_args(['9'])
Namespace(foo=9)
```

See the choices section for more details.

15.4.3.7. choices

Some command-line arguments should be selected from a restricted set of values. These can be handled by passing a container object as the choices keyword argument to add_argument(). When the command line is parsed, argument values will be checked, and an error message will be displayed if the argument was not one of the
acceptable values:

```python
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
```

usage: game.py [-h] {rock,paper,scissors}

```
usage:
game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock', 'paper', 'scissors')
```

Note that inclusion in the `choices` container is checked after any type conversions have been performed, so the type of the objects in the `choices` container should match the type specified:

```python
>>> parser = argparse.ArgumentParser(prog='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
```

usage: doors.py [-h] {1,2,3}
```
usage:
doors.py [-h] {1,2,3}
doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)
```

Any object that supports the `in` operator can be passed as the `choices` value, so `dict` objects, `set` objects, custom containers, etc. are all supported.

### 15.4.3.8. required

In general, the `argparse` module assumes that flags like `-f` and `--bar` indicate optional arguments, which can always be omitted at the command line. To make an option required, `True` can be specified for the `required` keyword argument to `add_argument()`:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
```
As the example shows, if an option is marked as required, `parse_args()` will report an error if that option is not present at the command line.

**Note:** Required options are generally considered bad form because users expect options to be optional, and thus they should be avoided when possible.

### 15.4.3.9. help

The `help` value is a string containing a brief description of the argument. When a user requests help (usually by using `-h` or `--help` at the command line), these `help` descriptions will be displayed with each argument:

```python
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', action='store_true',
... help='foo the bars before frobbing')
>>> parser.add_argument('bar', nargs='+',
... help='one of the bars to be frobbled')
>>> parser.parse_args(['-h'])
usage: frobble [-h] [--foo] bar [bar ...]
```

 positional arguments:
 bar one of the bars to be frobbled

 optional arguments:
 -h, --help show this help message and exit
 --foo foo the bars before frobbing

The `help` strings can include various format specifiers to avoid repetition of things like the program name or the argument default. The available specifiers include the program name, `%prog` and
most keyword arguments to add_argument(), e.g. %(default)s, % (type)s, etc.:

```python
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
help='the bar to %prog (default: %d
...)
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
  bar   the bar to frobble (default: 42)

optional arguments:
  -h, --help  show this help message and exit
```

argparse supports silencing the help entry for certain options, by setting the help value to argparse.SUPPRESS:

```python
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

optional arguments:
  -h, --help  show this help message and exit
```

15.4.3.10. metavar

When ArgumentParser generates help messages, it needs some way to refer to each expected argument. By default, ArgumentParser objects use the dest value as the “name” of each object. By default, for positional argument actions, the dest value is used directly, and for optional argument actions, the dest value is uppercased. So, a single positional argument with dest='bar' will be referred to as bar. A single optional argument --foo that should be followed by a single command-line argument will be referred to as foo. An example:
An alternative name can be specified with `metavar`:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage: [-h] [--foo YYY] bar

positional arguments:
  bar

optional arguments:
  -h, --help    show this help message and exit
  --foo YYY
```

Note that `metavar` only changes the displayed name - the name of the attribute on the `parse_args()` object is still determined by the `dest` value.

Different values of `nargs` may cause the metavar to be used multiple times. Providing a tuple to `metavar` specifies a different display for each of the arguments:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
```
15.4.3.11. dest

Most `ArgumentParser` actions add some value as an attribute of the object returned by `parse_args()`. The name of this attribute is determined by the `dest` keyword argument of `add_argument()`. For positional argument actions, `dest` is normally supplied as the first argument to `add_argument()`:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

For optional argument actions, the value of `dest` is normally inferred from the option strings. `ArgumentParser` generates the value of `dest` by taking the first long option string and stripping away the initial `--` string. If no long option strings were supplied, `dest` will be derived from the first short option string by stripping the initial `-` character. Any internal `-` characters will be converted to `_` characters to make sure the string is a valid attribute name. The examples below illustrate this behavior:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args(['-f 1 -x 2'.split()])
Namespace(foo_bar='1', x='2')
>>> parser.parse_args(['--foo 1 -y 2'.split()])
```
dest allows a custom attribute name to be provided:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

15.4.3.12. Action classes

Action classes implement the Action API, a callable which returns a callable which processes arguments from the command-line. Any object which follows this API may be passed as the action parameter to `add_argument()`.

```python
class argparse.Action(option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None)
```

Action objects are used by an ArgumentParser to represent the information needed to parse a single argument from one or more strings from the command line. The Action class must accept the two positional arguments plus any keyword arguments passed to `ArgumentParser.add_argument()` except for the action itself.

Instances of Action (or return value of any callable to the action parameter) should have attributes “dest”, “option_strings”, “default”, “type”, “required”, “help”, etc. defined. The easiest way to ensure these attributes are defined is to call `Action.__init__`.

Action instances should be callable, so subclasses must override the `__call__` method, which should accept four parameters:

- `parser` - The ArgumentParser object which contains this action.
- `namespace` - The `Namespace` object that will be returned by
parse_args(). Most actions add an attribute to this object using setattr().

- **values** - The associated command-line arguments, with any type conversions applied. Type conversions are specified with the type keyword argument to add_argument().

- **option_string** - The option string that was used to invoke this action. The option_string argument is optional, and will be absent if the action is associated with a positional argument.

The __call__ method may perform arbitrary actions, but will typically set attributes on the namespace based on dest and values.
15.4.4. The parse_args() method

ArgumentParser.parse_args(args=None, namespace=None)

Convert argument strings to objects and assign them as attributes of the namespace. Return the populated namespace.

Previous calls to add_argument() determine exactly what objects are created and how they are assigned. See the documentation for add_argument() for details.

By default, the argument strings are taken from sys.argv, and a new empty Namespace object is created for the attributes.

15.4.4.1. Option value syntax

The parse_args() method supports several ways of specifying the value of an option (if it takes one). In the simplest case, the option and its value are passed as two separate arguments:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

For long options (options with names longer than a single character), the option and value can also be passed as a single command-line argument, using = to separate them:

```python
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

For short options (options only one character long), the option and its
value can be concatenated:

```python
g>>> parser.parse_args(['-xx'])
Namespace(foo=None, x='X')
```

Several short options can be joined together, using only a single `-` prefix, as long as only the last option (or none of them) requires a value:

```python
g>>> parser = argparse.ArgumentParser(prog='PROG')
g>>> parser.add_argument('-x', action='store_true')
g>>> parser.add_argument('-y', action='store_true')
g>>> parser.add_argument('-z')
g>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

## 15.4.4.2. Invalid arguments

While parsing the command line, `parse_args()` checks for a variety of errors, including ambiguous options, invalid types, invalid options, wrong number of positional arguments, etc. When it encounters such an error, it exits and prints the error along with a usage message:

```python
g>>> parser = argparse.ArgumentParser(prog='PROG')
g>>> parser.add_argument('--foo', type=int)
g>>> parser.add_argument('bar', nargs='?')

# invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

# invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

# wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
15.4.4.3. Arguments containing -

The `parse_args()` method attempts to give errors whenever the user has clearly made a mistake, but some situations are inherently ambiguous. For example, the command-line argument `-1` could either be an attempt to specify an option or an attempt to provide a positional argument. The `parse_args()` method is cautious here: positional arguments may only begin with `-` if they look like negative numbers and there are no options in the parser that look like negative numbers:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```
If you have positional arguments that must begin with - and don’t look like negative numbers, you can insert the pseudo-argument '--' which tells `parse_args()` that everything after that is a positional argument:

```python
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

### 15.4.4.4. Argument abbreviations (prefix matching)

The `parse_args()` method allows long options to be abbreviated to a prefix, if the abbreviation is unambiguous (the prefix matches a unique option):

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

An error is produced for arguments that could produce more than one options.

### 15.4.4.5. Beyond `sys.argv`

Sometimes it may be useful to have an ArgumentParser parse arguments other than those of `sys.argv`. This can be accomplished by passing a list of strings to `parse_args()`. This is useful for testing at the interactive prompt:

```python
>>> parser = argparse.ArgumentParser()
```
15.4.4.6. The Namespace object

```
>>> parser.add_argument(  
    ...     'integers', metavar='int', type=int, choices=xrange(10),  
    ...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(  
    ...     '--sum', dest='accumulate', action='store_const', const=max,  
    ...     default=max, help='sum the integers (default: find the max)
```

```
>>> parser.parse_args([1, 2, 3])  
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3])
>>> parser.parse_args([1, 2, 3, '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3])
```

```
15.4.4.6. The Namespace object

class argparse.Namespace

    Simple class used by default by parse_args() to create an object  
    holding attributes and return it.

This class is deliberately simple, just an object subclass with a  
readable string representation. If you prefer to have dict-like view  
of the attributes, you can use the standard Python idiom, vars():

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> vars(parser.parse_args(['--foo', 'BAR']))
{'foo': 'BAR'}
```

It may also be useful to have an ArgumentParser assign attributes to  
an already existing object, rather than a new Namespace object. This  
can be achieved by specifying the namespace= keyword argument:

```
>>> class C(object):
    ...    pass
    ...
>>> c = C()
>>> parser = argparse.ArgumentParser('--foo')
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
```
15.4.5. Other utilities

15.4.5.1. Sub-commands

ArgumentParser.\texttt{add\_subparsers([title][, description][, prog][, parser\_class][, action][, option\_string][, dest][, help][, metavar])}

Many programs split up their functionality into a number of subcommands, for example, the svn program can invoke subcommands like \texttt{svn checkout}, \texttt{svn update}, and \texttt{svn commit}. Splitting up functionality this way can be a particularly good idea when a program performs several different functions which require different kinds of command-line arguments. \texttt{ArgumentParser} supports the creation of such sub-commands with the \texttt{add\_subparsers()} method. The \texttt{add\_subparsers()} method is normally called with no arguments and returns a special action object. This object has a single method, \texttt{add\_parser()}, which takes a command name and any \texttt{ArgumentParser} constructor arguments, and returns an \texttt{ArgumentParser} object that can be modified as usual.

Description of parameters:

- \texttt{title} - title for the sub-parser group in help output; by default “subcommands” if description is provided, otherwise uses title for positional arguments
- \texttt{description} - description for the sub-parser group in help output, by default \texttt{None}
- \texttt{prog} - usage information that will be displayed with subcommand help, by default the name of the program and any positional arguments before the subparser argument
- \texttt{parser\_class} - class which will be used to create sub-parser
instances, by default the class of the current parser (e.g. ArgumentParser)

- **action** - the basic type of action to be taken when this argument is encountered at the command line
- **dest** - name of the attribute under which sub-command name will be stored; by default `None` and no value is stored
- **help** - help for sub-parser group in help output, by default `None`
- **metavar** - string presenting available sub-commands in help; by default it is `None` and presents sub-commands in form `{cmd1, cmd2, ..}`

Some example usage:

```python
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='sub-command help')

>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')

>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices='XYZ', help='baz help')

>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

Note that the object returned by `parse_args()` will only contain attributes for the main parser and the subparser that was selected by the command line (and not any other subparsers). So in the example above, when the `a` command is specified, only the `foo` and `bar` attributes are present, and when the `b` command is
specified, only the foo and baz attributes are present.

Similarly, when a help message is requested from a subparser, only the help for that particular parser will be printed. The help message will not include parent parser or sibling parser messages. (A help message for each subparser command, however, can be given by supplying the help= argument to add_parser() as above.)

```python
>>> parser.parse_args(["--help")
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}  sub-command help
     a  a help
     b  b help

optional arguments:
  -h, --help  show this help message and exit
  --foo  foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar  bar help

optional arguments:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

optional arguments:
  -h, --help  show this help message and exit
  --baz {X,Y,Z}  baz help
```

The add_subparsers() method also supports title and description keyword arguments. When either is present, the subparser’s commands will appear in their own group in the help output. For example:
One particularly effective way of handling sub-commands is to combine the use of the `add_subparsers()` method with calls to `set_defaults()` so that each subparser knows which Python function it should execute. For example:

```python
>>> # sub-command functions
>>> def foo(args):
...     print args.x * args.y
...     ...
>>> def bar(args):
...     print '(%s)' % args.z
...     
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()

>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)

>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
```
This way, you can let `parse_args()` do the job of calling the appropriate function after argument parsing is complete. Associating functions with actions like this is typically the easiest way to handle the different actions for each of your subparsers. However, if it is necessary to check the name of the subparser that was invoked, the `dest` keyword argument to the `add_subparsers()` call will work:

```python
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

## 15.4.5.2. FileType objects

```python
class argparse.FileType(mode='r', bufsize=None)
```

The `FileType` factory creates objects that can be passed to the type argument of `ArgumentParser.add_argument()`. Arguments that have `FileType` objects as their type will open command-line arguments as files with the requested modes and buffer sizes:

```python
>>> parser = argparse.ArgumentParser()
```
FileType objects understand the pseudo-argument `--output` and automatically convert this into `sys.stdin` for readable FileType objects and `sys.stdout` for writable FileType objects:

```
>>> parser.add_argument('--output', type=argparse.FileType('wb'))
>>> parser.parse_args(['--output', 'out'])
Namespace(output=<open file 'out', mode 'wb' at 0x...>)
```

## 15.4.5.3. Argument groups

`ArgumentParser.add_argument_group(title=None, description=None)`

By default, `ArgumentParser` groups command-line arguments into “positional arguments” and “optional arguments” when displaying help messages. When there is a better conceptual grouping of arguments than this default one, appropriate groups can be created using the `add_argument_group()` method:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
    bar     bar help
    --foo FOO  foo help
```

The `add_argument_group()` method returns an argument group object which has an `add_argument()` method just like a regular
**ArgumentParser.** When an argument is added to the group, the parser treats it just like a normal argument, but displays the argument in a separate group for help messages. The `add_argument_group()` method accepts *title* and *description* arguments which can be used to customize this display:

```python
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [-bar BAR] foo

    group1:
    group1 description
    
    foo foo help

    group2:
    group2 description
    
    --bar BAR bar help
```

Note that any arguments not in your user-defined groups will end up back in the usual “positional arguments” and “optional arguments” sections.

### 15.4.5.4. Mutual exclusion

**ArgumentParser.** `.add_mutually_exclusive_group(required=False)`

Create a mutually exclusive group. *argparse* will make sure that only one of the arguments in the mutually exclusive group was present on the command line:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
```
The **add_mutually_exclusive_group()** method also accepts a `required` argument, to indicate that at least one of the mutually exclusive arguments is required:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

Note that currently mutually exclusive argument groups do not support the `title` and `description` arguments of `add_argument_group()`.

### 15.4.5.5. Parser defaults

**ArgumentParser.set_defaults(**kwargs)**

Most of the time, the attributes of the object returned by `parse_args()` will be fully determined by inspecting the command-line arguments and the argument actions. `set_defaults()` allows some additional attributes that are determined without any inspection of the command line to be added:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
```
Note that parser-level defaults always override argument-level defaults:

```python
g>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

Parser-level defaults can be particularly useful when working with multiple parsers. See the `add_subparsers()` method for an example of this type.

```
ArgumentParser.get_default(dest)
Get the default value for a namespace attribute, as set by either add_argument() or by set_defaults():
```

```python
g>>> parser = argparse.ArgumentParser()
g>>> parser.add_argument('--foo', default='bar')
g>>> parser.set_defaults(foo='spam')
g>>> parser.parse_args([])
Namespace(foo='spam')
```

15.4.5.6. Printing help

In most typical applications, `parse_args()` will take care of formatting and printing any usage or error messages. However, several formatting methods are available:

```
ArgumentParser.print_usage(file=None)
Print a brief description of how the ArgumentParser should be invoked on the command line. If file is None, sys.stdout is assumed.
```

```
ArgumentParser.print_help(file=None)
```
Print a help message, including the program usage and information about the arguments registered with the ArgumentParser. If file is None, sys.stdout is assumed.

There are also variants of these methods that simply return a string instead of printing it:

ArgumentParser.format_usage()
Return a string containing a brief description of how the ArgumentParser should be invoked on the command line.

ArgumentParser.format_help()
Return a string containing a help message, including the program usage and information about the arguments registered with the ArgumentParser.

15.4.5.7. Partial parsing

ArgumentParser.parse_known_args(args=None, namespace=None)

Sometimes a script may only parse a few of the command-line arguments, passing the remaining arguments on to another script or program. In these cases, the parse_known_args() method can be useful. It works much like parse_args() except that it does not produce an error when extra arguments are present. Instead, it returns a two item tuple containing the populated namespace and the list of remaining argument strings.

>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])

Warning: Prefix matching rules apply to parse_known_args(). The
15.4.5.8. Customizing file parsing

`ArgumentParser.convert_arg_line_to_args(arg_line)`

Arguments that are read from a file (see the `fromfile_prefix_chars` keyword argument to the `ArgumentParser` constructor) are read one argument per line. `convert_arg_line_to_args()` can be overridden for fancier reading.

This method takes a single argument `arg_line` which is a string read from the argument file. It returns a list of arguments parsed from this string. The method is called once per line read from the argument file, in order.

A useful override of this method is one that treats each space-separated word as an argument:

```python
def convert_arg_line_to_args(self, arg_line):
    return arg_line.split()
```

15.4.5.9. Exiting methods

`ArgumentParser.exit(status=0, message=None)`

This method terminates the program, exiting with the specified `status` and, if given, it prints a `message` before that.

`ArgumentParser.error(message)`

This method prints a usage message including the `message` to the standard error and terminates the program with a status code of 2.
15.4.6. Upgrading optparse code

Originally, the `argparse` module had attempted to maintain compatibility with `optparse`. However, `optparse` was difficult to extend transparently, particularly with the changes required to support the new `nargs=` specifiers and better usage messages. When most everything in `optparse` had either been copy-pasted over or monkey-patched, it no longer seemed practical to try to maintain the backwards compatibility.

The `argparse` module improves on the standard library `optparse` module in a number of ways including:

- Handling positional arguments.
- Supporting sub-commands.
- Allowing alternative option prefixes like `+` and `/`.
- Handling zero-or-more and one-or-more style arguments.
- Producing more informative usage messages.
- Providing a much simpler interface for custom `type` and `action`.

A partial upgrade path from `optparse` to `argparse`:

- Replace all `optparse.OptionParser.add_option()` calls with `ArgumentParser.add_argument()` calls.
- Replace `(options, args) = parser.parse_args()` with `args = parser.parse_args()` and add additional `ArgumentParser.add_argument()` calls for the positional arguments. Keep in mind that what was previously called `options`, now in `argparse` context is called `args`.
- Replace callback actions and the `callback_*` keyword arguments with `type` or `action` arguments.
- Replace string names for `type` keyword arguments with the
corresponding type objects (e.g. int, float, complex, etc).


- Replace strings with implicit arguments such as `%default` or `%prog` with the standard Python syntax to use dictionaries to format strings, that is, `%{default}s` and `%{prog}s`.

- Replace the OptionParser constructor `version` argument with a call to `parser.add_argument('--version', action='version', version='<the version>')`. 
15. Generic Operating System Services
New in version 2.3.

Deprecated since version 2.7: The optparse module is deprecated and will not be developed further; development will continue with the argparse module.

Source code: Lib/optparse.py

optparse is a more convenient, flexible, and powerful library for parsing command-line options than the old getopt module. optparse uses a more declarative style of command-line parsing: you create an instance of OptionParser, populate it with options, and parse the command line. optparse allows users to specify options in the conventional GNU/POSIX syntax, and additionally generates usage and help messages for you.

Here's an example of using optparse in a simple script:

```python
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
  action="store_false", dest="verbose",
  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```
With these few lines of code, users of your script can now do the “usual thing” on the command-line, for example:

```
<yourscript> --file=outfile -q
```

As it parses the command line, `optparse` sets attributes of the `options` object returned by `parse_args()` based on user-supplied command-line values. When `parse_args()` returns from parsing this command line, `options.filename` will be “outfile” and `options.verbose` will be False. `optparse` supports both long and short options, allows short options to be merged together, and allows options to be associated with their arguments in a variety of ways. Thus, the following command lines are all equivalent to the above example:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -f outfile
<yourscript> -qf outfile
```

Additionally, users can run one of

```
<yourscript> -h
<yourscript> --help
```

and `optparse` will print out a brief summary of your script’s options:

```
Usage: <yourscript> [options]

Options:
  -h, --help          show this help message and exit
  -f FILE, --file=FILE write report to FILE
  -q, --quiet         don't print status messages to stdout
```

where the value of `yourscript` is determined at runtime (normally from `sys.argv[0]`).
15.5.1. Background

`optparse` was explicitly designed to encourage the creation of programs with straightforward, conventional command-line interfaces. To that end, it supports only the most common command-line syntax and semantics conventionally used under Unix. If you are unfamiliar with these conventions, read this section to acquaint yourself with them.

15.5.1.1. Terminology

**argument**

A string entered on the command-line, and passed by the shell to `exec1()` or `execv()`. In Python, arguments are elements of `sys.argv[1:]` (`sys.argv[0]` is the name of the program being executed). Unix shells also use the term “word”.

It is occasionally desirable to substitute an argument list other than `sys.argv[1:]`, so you should read “argument” as “an element of `sys.argv[1:]`, or of some other list provided as a substitute for `sys.argv[1:]`”.

**option**

An argument used to supply extra information to guide or customize the execution of a program. There are many different syntaxes for options; the traditional Unix syntax is a hyphen ("-") followed by a single letter, e.g. `-x` or `-F`. Also, traditional Unix syntax allows multiple options to be merged into a single argument, e.g. `-x -F` is equivalent to `-xF`. The GNU project introduced `--` followed by a series of hyphen-separated words, e.g. `--file` or `--dry-run`. These are the only two option syntaxes provided by `optparse`. 
Some other option syntaxes that the world has seen include:

- a hyphen followed by a few letters, e.g. `-pf` (this is *not* the same as multiple options merged into a single argument)
- a hyphen followed by a whole word, e.g. `-file` (this is technically equivalent to the previous syntax, but they aren’t usually seen in the same program)
- a plus sign followed by a single letter, or a few letters, or a word, e.g. `+f`, `+rgb`
- a slash followed by a letter, or a few letters, or a word, e.g. `/f`, `/file`

These option syntaxes are not supported by `optparse`, and they never will be. This is deliberate: the first three are non-standard on any environment, and the last only makes sense if you’re exclusively targeting VMS, MS-DOS, and/or Windows.

**option argument**

an argument that follows an option, is closely associated with that option, and is consumed from the argument list when that option is. With `optparse`, option arguments may either be in a separate argument from their option:

```
-f foo
--file foo
```

or included in the same argument:

```
-ffoo
--file=foo
```

Typically, a given option either takes an argument or it doesn’t. Lots of people want an “optional option arguments” feature, meaning that some options will take an argument if they see it, and won’t if they don’t. This is somewhat controversial, because it makes parsing ambiguous: if `-a` takes an optional argument and
-b is another option entirely, how do we interpret -ab? Because of this ambiguity, optparse does not support this feature.

Positional argument
something leftover in the argument list after options have been parsed, i.e. after options and their arguments have been parsed and removed from the argument list.

Required option
an option that must be supplied on the command-line; note that the phrase “required option” is self-contradictory in English. optparse doesn’t prevent you from implementing required options, but doesn’t give you much help at it either.

For example, consider this hypothetical command-line:

```
prog -v --report report.txt foo bar
```

-v and --report are both options. Assuming that --report takes one argument, report.txt is an option argument. foo and bar are positional arguments.

15.5.1.2. What are options for?

Options are used to provide extra information to tune or customize the execution of a program. In case it wasn’t clear, options are usually optional. A program should be able to run just fine with no options whatsoever. (Pick a random program from the Unix or GNU toolsets. Can it run without any options at all and still make sense? The main exceptions are find, tar, and dd—all of which are mutant oddballs that have been rightly criticized for their non-standard syntax and confusing interfaces.)

Lots of people want their programs to have “required options”. Think about it. If it’s required, then it’s not optional! If there is a piece of
information that your program absolutely requires in order to run successfully, that’s what positional arguments are for.

As an example of good command-line interface design, consider the humble `cp` utility, for copying files. It doesn't make much sense to try to copy files without supplying a destination and at least one source. Hence, `cp` fails if you run it with no arguments. However, it has a flexible, useful syntax that does not require any options at all:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

You can get pretty far with just that. Most `cp` implementations provide a bunch of options to tweak exactly how the files are copied: you can preserve mode and modification time, avoid following symlinks, ask before clobbering existing files, etc. But none of this distracts from the core mission of `cp`, which is to copy either one file to another, or several files to another directory.

### 15.5.1.3. What are positional arguments for?

Positional arguments are for those pieces of information that your program absolutely, positively requires to run.

A good user interface should have as few absolute requirements as possible. If your program requires 17 distinct pieces of information in order to run successfully, it doesn't much matter how you get that information from the user—most people will give up and walk away before they successfully run the program. This applies whether the user interface is a command-line, a configuration file, or a GUI: if you make that many demands on your users, most of them will simply give up.

In short, try to minimize the amount of information that users are absolutely required to supply—use sensible defaults whenever
possible. Of course, you also want to make your programs reasonably flexible. That's what options are for. Again, it doesn't matter if they are entries in a config file, widgets in the “Preferences” dialog of a GUI, or command-line options—the more options you implement, the more flexible your program is, and the more complicated its implementation becomes. Too much flexibility has drawbacks as well, of course; too many options can overwhelm users and make your code much harder to maintain.
While `optparse` is quite flexible and powerful, it’s also straightforward to use in most cases. This section covers the code patterns that are common to any `optparse`-based program.

First, you need to import the OptionParser class; then, early in the main program, create an OptionParser instance:

```python
from optparse import OptionParser
...
parser = OptionParser()
```

Then you can start defining options. The basic syntax is:

```python
parser.add_option(opt_str, ..., attr=value, ...)
```

Each option has one or more option strings, such as `-f` or `--file`, and several option attributes that tell `optparse` what to expect and what to do when it encounters that option on the command line.

Typically, each option will have one short option string and one long option string, e.g.:

```python
parser.add_option("-f", "--file", ...)
```

You’re free to define as many short option strings and as many long option strings as you like (including zero), as long as there is at least one option string overall.

The option strings passed to `OptionParser.add_option()` are effectively labels for the option defined by that call. For brevity, we will frequently refer to *encountering an option* on the command line;
in reality, `optparse` encounters `option strings` and looks up options from them.

Once all of your options are defined, instruct `optparse` to parse your program’s command line:

```python
(options, args) = parser.parse_args()
```

(If you like, you can pass a custom argument list to `parse_args()`, but that's rarely necessary: by default it uses `sys.argv[1:]`.)

`parse_args()` returns two values:

- `options`, an object containing values for all of your options—e.g. if `--file` takes a single string argument, then `options.file` will be the filename supplied by the user, or `None` if the user did not supply that option
- `args`, the list of positional arguments leftover after parsing options

This tutorial section only covers the four most important option attributes: `action`, `type`, `dest` (destination), and `help`. Of these, `action` is the most fundamental.

### 15.5.2.1. Understanding option actions

Actions tell `optparse` what to do when it encounters an option on the command line. There is a fixed set of actions hard-coded into `optparse`; adding new actions is an advanced topic covered in section `Extending optparse`. Most actions tell `optparse` to store a value in some variable—for example, take a string from the command line and store it in an attribute of `options`.

If you don’t specify an option action, `optparse` defaults to `store`.
15.5.2.2. The store action

The most common option action is `store`, which tells `optparse` to take the next argument (or the remainder of the current argument), ensure that it is of the correct type, and store it to your chosen destination.

For example:

```python
parser.add_option("-f", "--file",
action="store", type="string", dest="filename"
```

Now let's make up a fake command line and ask `optparse` to parse it:

```python
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

When `optparse` sees the option string `-f`, it consumes the next argument, `foo.txt`, and stores it in `options.filename`. So, after this call to `parse_args()`, `options.filename` is `"foo.txt"`.

Some other option types supported by `optparse` are `int` and `float`. Here's an option that expects an integer argument:

```python
parser.add_option("-n", type="int", dest="num")
```

Note that this option has no long option string, which is perfectly acceptable. Also, there's no explicit action, since the default is `store`.

Let's parse another fake command line. This time, we'll jam the option argument right up against the option: since `-n42` (one argument) is equivalent to `-n 42` (two arguments), the code
(options, args) = parser.parse_args(['-n42'])
print options.num

will print 42.

If you don’t specify a type, `optparse` assumes `string`. Combined with the fact that the default action is `store`, that means our first example can be a lot shorter:

```python
parser.add_option("-f", "--file", dest="filename")
```

If you don’t supply a destination, `optparse` figures out a sensible default from the option strings: if the first long option string is `--foo-bar`, then the default destination is `foo_bar`. If there are no long option strings, `optparse` looks at the first short option string: the default destination for `-f` is `f`.

`optparse` also includes built-in `long` and `complex` types. Adding types is covered in section `Extending optparse`.

### 15.5.2.3. Handling boolean (flag) options

Flag options—set a variable to true or false when a particular option is seen—are quite common. `optparse` supports them with two separate actions, `store_true` and `store_false`. For example, you might have a `verbose` flag that is turned on with `-v` and off with `-q`:

```python
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Here we have two different options with the same destination, which is perfectly OK. (It just means you have to be a bit careful when setting default values—see below.)
When `optparse` encounters `-v` on the command line, it sets `options.verbose` to `True`; when it encounters `-q`, `options.verbose` is set to `False`.

### 15.5.2.4. Other actions

Some other actions supported by `optparse` are:

- "store_const"
  - store a constant value
- "append"
  - append this option's argument to a list
- "count"
  - increment a counter by one
- "callback"
  - call a specified function

These are covered in section Reference Guide, Reference Guide and section Option Callbacks.

### 15.5.2.5. Default values

All of the above examples involve setting some variable (the "destination") when certain command-line options are seen. What happens if those options are never seen? Since we didn’t supply any defaults, they are all set to `None`. This is usually fine, but sometimes you want more control. `optparse` lets you supply a default value for each destination, which is assigned before the command line is parsed.

First, consider the verbose/quiet example. If we want `optparse` to set `verbose` to `True` unless `-q` is seen, then we can do this:

```python
parser.add_option("-v", action="store_true", dest="verbose", de
Since default values apply to the destination rather than to any particular option, and these two options happen to have the same destination, this is exactly equivalent:

```python
parser.add_option("-q", action="store_false", dest="verbose")
parser.add_option("-v", action="store_true", dest="verbose")
```

Consider this:

```python
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Again, the default value for verbose will be True: the last default value supplied for any particular destination is the one that counts.

A clearer way to specify default values is the set_defaults() method of OptionParser, which you can call at any time before calling parse_args():

```python
parser.set_defaults(verbosetrue=True)
pclassifier.add_option(...) (options, args) = parser.parse_args()
```

As before, the last value specified for a given option destination is the one that counts. For clarity, try to use one method or the other of setting default values, not both.

### 15.5.2.6. Generating help

`optparse`'s ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a help value for each option, and optionally a
short usage message for your whole program. Here's an OptionParser populated with user-friendly (documented) options:

```python
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default="make lots of noise [default]"
)
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)"
)
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE"
)
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, or expert [default: %default]"
)
```

If `optparse` encounters either `-h` or `--help` on the command-line, or if you just call `parser.print_help()`, it prints the following to standard output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
-h, --help show this help message and exit
-v, --verbose make lots of noise [default]
-q, --quiet be vewwy quiet (I'm hunting wabbits)
-f FILE, --filename=FILE write output to FILE
-m MODE, --mode=MODE interaction mode: novice, intermediate, expert [default: intermediate]
```

(If the help output is triggered by a help option, `optparse` exits after printing the help text.)

There's a lot going on here to help `optparse` generate the best possible help message:
- the script defines its own usage message:

```
usage = "usage: %prog [options] arg1 arg2"
```

`optparse` expands `%prog` in the usage string to the name of the current program, i.e. `os.path.basename(sys.argv[0])`. The expanded string is then printed before the detailed option help.

If you don’t supply a usage string, `optparse` uses a bland but sensible default: "Usage: %prog [options]", which is fine if your script doesn’t take any positional arguments.

- every option defines a help string, and doesn’t worry about line-wrapping—`optparse` takes care of wrapping lines and making the help output look good.

- options that take a value indicate this fact in their automatically-generated help message, e.g. for the “mode” option:

```
-m MODE, --mode=MODE
```

Here, “MODE” is called the meta-variable: it stands for the argument that the user is expected to supply to `-m/--mode`. By default, `optparse` converts the destination variable name to uppercase and uses that for the meta-variable. Sometimes, that’s not what you want—for example, the `--filename` option explicitly sets `metavar="FILE"`, resulting in this automatically-generated option description:

```
-f FILE, --filename=FILE
```

This is important for more than just saving space, though: the manually written help text uses the meta-variable `FILE` to clue the user in that there’s a connection between the semi-formal
syntax -f FILE and the informal semantic description “write output to FILE”. This is a simple but effective way to make your help text a lot clearer and more useful for end users.

New in version 2.4: Options that have a default value can include %default in the help string—optparse will replace it with str() of the option’s default value. If an option has no default value (or the default value is None), %default expands to none.

### 15.5.2.6.1. Grouping Options

When dealing with many options, it is convenient to group these options for better help output. An OptionParser can contain several option groups, each of which can contain several options.

An option group is obtained using the class OptionGroup:

```python
class optparse.**OptionGroup**(parser, title, description=None)
```

where

- parser is the OptionParser instance the group will be inserted in to
- title is the group title
- description, optional, is a long description of the group

OptionGroup inherits from OptionContainer (like OptionParser) and so the add_option() method can be used to add an option to the group.

Once all the options are declared, using the OptionParser method add_option_group() the group is added to the previously defined parser.

Continuing with the parser defined in the previous section, adding an OptionGroup to a parser is easy:
This would result in the following help output:

Usage: <yourscript> [options] arg1 arg2

Options:
- h, --help show this help message and exit
- v, --verbose make lots of noise [default]
- q, --quiet be vewwy quiet (I'm hunting wabbits)
- f FILE, --filename=FILE write output to FILE
- m MODE, --mode=MODE interaction mode: novice, intermediate, expert [default: intermediate]

Dangerous Options:
Caution: use these options at your own risk. It is believed that some of them bite.

-g Group option.

A bit more complete example might involve using more than one group: still extending the previous example:
that results in the following output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help          show this help message and exit
  -v, --verbose       make lots of noise [default]
  -q, --quiet         be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE write output to FILE
  -m MODE, --mode=MODE interaction mode: novice, intermediate, [default: intermediate]

Dangerous Options:
  Caution: use these options at your own risk. It is believe of them bite.
  -g

Debug Options:
  -d, --debug         Print debug information
  -s, --sql           Print all SQL statements executed
  -e
```

Another interesting method, in particular when working programmatically with option groups is:

```
OptionParser.get_option_group(opt_str)
```

Return the `OptionGroup` to which the short or long option string `opt_str` (e.g. `-o` or `--option`) belongs. If there's no such `OptionGroup`, return `None`.

### 15.5.2.7. Printing a version string

Similar to the brief usage string, `optparse` can also print a version string for your program. You have to supply the string as the `version` argument to OptionParser:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```
%prog is expanded just like it is in usage. Apart from that, version can contain anything you like. When you supply it, optparse automatically adds a --version option to your parser. If it encounters this option on the command line, it expands your version string (by replacing %prog), prints it to stdout, and exits.

For example, if your script is called /usr/bin/foo:

```
$ /usr/bin/foo --version
foo 1.0
```

The following two methods can be used to print and get the version string:

OptionParser. **print_version**(file=None)
   
   Print the version message for the current program (self.version) to file (default stdout). As with print_usage(), any occurrence of %prog in self.version is replaced with the name of the current program. Does nothing if self.version is empty or undefined.

OptionParser. **get_version**()
   
   Same as print_version() but returns the version string instead of printing it.

### 15.5.2.8. How optparse handles errors

There are two broad classes of errors that optparse has to worry about: programmer errors and user errors. Programmer errors are usually erroneous calls to OptionParser.add_option(), e.g. invalid option strings, unknown option attributes, missing option attributes, etc. These are dealt with in the usual way: raise an exception (either optparse.OptionError or TypeError) and let the program crash.
Handling user errors is much more important, since they are guaranteed to happen no matter how stable your code is. `optparse` can automatically detect some user errors, such as bad option arguments (passing `-n 4x` where `-n` takes an integer argument), missing arguments (`-n` at the end of the command line, where `-n` takes an argument of any type). Also, you can call `OptionParser.error()` to signal an application-defined error condition:

```python
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

In either case, `optparse` handles the error the same way: it prints the program’s usage message and an error message to standard error and exits with error status 2.

Consider the first example above, where the user passes `4x` to an option that takes an integer:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]
foo: error: option -n: invalid integer value: '4x'
```

Or, where the user fails to pass a value at all:

```
$ /usr/bin/foo -n
Usage: foo [options]
foo: error: -n option requires an argument
```

`optparse`-generated error messages take care always to mention the option involved in the error; be sure to do the same when calling `OptionParser.error()` from your application code.
If `optparse`'s default error-handling behaviour does not suit your needs, you'll need to subclass `OptionParser` and override its `exit()` and/or `error()` methods.

### 15.5.2.9. Putting it all together

Here's what `optparse`-based scripts usually look like:

```python
from optparse import OptionParser
...

def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    ...
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print "reading %s..." % options.filename
    ...

if __name__ == "__main__":
    main()
```
15.5.3. Reference Guide

15.5.3.1. Creating the parser

The first step in using optparse is to create an OptionParser instance.

```python
class optparse.OptionParser(...)
```

The OptionParser constructor has no required arguments, but a number of optional keyword arguments. You should always pass them as keyword arguments, i.e. do not rely on the order in which the arguments are declared.

```python
usage (default: "%prog [options]")
```

The usage summary to print when your program is run incorrectly or with a help option. When optparse prints the usage string, it expands `%prog` to `os.path.basename(sys.argv[0])` (or to `prog` if you passed that keyword argument). To suppress a usage message, pass the special value `optparse.SUPPRESS_USAGE`.

```python
option_list (default: [])
```

A list of Option objects to populate the parser with. The options in `option_list` are added after any options in `standard_option_list` (a class attribute that may be set by OptionParser subclasses), but before any version or help options. Deprecated; use `add_option()` after creating the parser instead.

```python
option_class (default: optparse.Option)
```

Class to use when adding options to the parser in `add_option()`.
version (default: None)

A version string to print when the user supplies a version option. If you supply a true value for `version`, `optparse` automatically adds a version option with the single option string `-version`. The substring `%prog` is expanded the same as for `usage`.

conflict_handler (default: "error")

Specifies what to do when options with conflicting option strings are added to the parser; see section Conflicts between options.

description (default: None)

A paragraph of text giving a brief overview of your program. `optparse` reformats this paragraph to fit the current terminal width and prints it when the user requests help (after `usage`, but before the list of options).

formatter (default: a new IndentedHelpFormatter)

An instance of `optparse.HelpFormatter` that will be used for printing help text. `optparse` provides two concrete classes for this purpose: IndentedHelpFormatter and TitledHelpFormatter.

add_help_option (default: True)

If true, `optparse` will add a help option (with option strings `-h` and `--help`) to the parser.

prog

The string to use when expanding `%prog` in `usage` and `version` instead of `os.path.basename(sys.argv[0])`.

epilog (default: None)

A paragraph of help text to print after the option help.
15.5.3.2. Populating the parser

There are several ways to populate the parser with options. The preferred way is by using `OptionParser.add_option()`, as shown in section Tutorial. `add_option()` can be called in one of two ways:

- pass it an Option instance (as returned by `make_option()`)
- pass it any combination of positional and keyword arguments that are acceptable to `make_option()` (i.e., to the Option constructor), and it will create the Option instance for you

The other alternative is to pass a list of pre-constructed Option instances to the OptionParser constructor, as in:

```python
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
]
parser = OptionParser(option_list=option_list)
```

(`make_option()` is a factory function for creating Option instances; currently it is an alias for the Option constructor. A future version of `optparse` may split Option into several classes, and `make_option()` will pick the right class to instantiate. Do not instantiate Option directly.)

15.5.3.3. Defining options

Each Option instance represents a set of synonymous command-line option strings, e.g. `-f` and `--file`. You can specify any number of short or long option strings, but you must specify at least one overall option string.
The canonical way to create an `Option` instance is with the `add_option()` method of `OptionParser`.

```python
OptionParser.add_option(option)
OptionParser.add_option(*opt_str, attr=value, ...)
```

To define an option with only a short option string:

```python
parser.add_option("-f", attr=value, ...)
```

And to define an option with only a long option string:

```python
parser.add_option("--foo", attr=value, ...)
```

The keyword arguments define attributes of the new Option object. The most important option attribute is `action`, and it largely determines which other attributes are relevant or required. If you pass irrelevant option attributes, or fail to pass required ones, `optparse` raises an `OptionError` exception explaining your mistake.

An option's `action` determines what `optparse` does when it encounters this option on the command-line. The standard option actions hard-coded into `optparse` are:

- "store"
  - store this option's argument (default)
- "store_const"
  - store a constant value
- "store_true"
  - store a true value
- "store_false"
  - store a false value
- "append"
append this option’s argument to a list

"append_const"

append a constant value to a list

"count"

increment a counter by one

"callback"

call a specified function

"help"

print a usage message including all options and the documentation for them

(If you don’t supply an action, the default is "store". For this action, you may also supply type and dest option attributes; see Standard option actions.)

As you can see, most actions involve storing or updating a value somewhere. optparse always creates a special object for this, conventionally called options (it happens to be an instance of optparse.Values). Option arguments (and various other values) are stored as attributes of this object, according to the dest (destination) option attribute.

For example, when you call

```
parser.parse_args()
```

one of the first things optparse does is create the options object:

```
options = Values()
```

If one of the options in this parser is defined with

```
parser.add_option("-f", "--file", action="store", type="string"
```
and the command-line being parsed includes any of the following:

- `ffoo`
- `-f foo`
- `--file=foo`
- `--file foo`

then `optparse`, on seeing this option, will do the equivalent of

```python
options.filename = "foo"
```

The `type` and `dest` option attributes are almost as important as `action`, but `action` is the only one that makes sense for all options.

### 15.5.3.4. Option attributes

The following option attributes may be passed as keyword arguments to `OptionParser.add_option()`. If you pass an option attribute that is not relevant to a particular option, or fail to pass a required option attribute, `optparse` raises `OptionError`.

**Option. action**

*(default: "store")*

Determines `optparse`'s behaviour when this option is seen on the command line; the available options are documented here.

**Option. type**

*(default: "string")*

The argument type expected by this option (e.g., "string" or "int"); the available option types are documented here.

**Option. dest**

*(default: derived from option strings)*
If the option’s action implies writing or modifying a value somewhere, this tells `optparse` where to write it: `dest` names an attribute of the `options` object that `optparse` builds as it parses the command line.

Option. **default**

The value to use for this option’s destination if the option is not seen on the command line. See also `OptionParser.set_defaults()`.

Option. **nargs**

(default: 1)

How many arguments of type `type` should be consumed when this option is seen. If > 1, `optparse` will store a tuple of values to `dest`.

Option. **const**

For actions that store a constant value, the constant value to store.

Option. **choices**

For options of type "choice", the list of strings the user may choose from.

Option. **callback**

For options with action "callback", the callable to call when this option is seen. See section Option Callbacks for detail on the arguments passed to the callable.

Option. **callback_args**

Option. **callback_kwargs**

Additional positional and keyword arguments to pass to `callback` after the four standard callback arguments.
Option: **help**

Help text to print for this option when listing all available options after the user supplies a **help** option (such as **--help**). If no help text is supplied, the option will be listed without help text. To hide this option, use the special value `optparse.SUPPRESS_HELP`.

Option: **metavar**

(default: derived from option strings)

Stand-in for the option argument(s) to use when printing help text. See section Tutorial for an example.

### 15.5.3.5. Standard option actions

The various option actions all have slightly different requirements and effects. Most actions have several relevant option attributes which you may specify to guide optparse's behaviour; a few have required attributes, which you must specify for any option using that action.

- **"store"** [relevant: `type`, `dest`, `nargs`, `choices`]

  The option must be followed by an argument, which is converted to a value according to `type` and stored in `dest`. If `nargs > 1`, multiple arguments will be consumed from the command line; all will be converted according to `type` and stored to `dest` as a tuple. See the Standard option types section.

  If `choices` is supplied (a list or tuple of strings), the type defaults to "choice".

  If `type` is not supplied, it defaults to "string".

  If `dest` is not supplied, optparse derives a destination from the first long option string (e.g., **--foo-bar** implies `foo_bar`). If there
are no long option strings, optparse derives a destination from the first short option string (e.g., -f implies f).

Example:

```python
parser.add_option("-f")
parser.add_option("-p", type="float", nargs=3, dest="point")
```

As it parses the command line

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

optparse will set

```python
options.f = "foo.txt"
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"
```

- **"store_const"** [required: const; relevant: dest]

  The value const is stored in dest.

  Example:

  ```python
  parser.add_option("-q", "--quiet",
                   action="store_const", const=0, dest="verb",
  parser.add_option("-v", "--verbose",
                   action="store_const", const=1, dest="verb",
  parser.add_option("--noisy",
                   action="store_const", const=2, dest="verb")
  ```

  If --noisy is seen, optparse will set

  ```python
  options.verbos = 2
  ```

- **"store_true"** [relevant: dest]
A special case of "store_const" that stores a true value to dest.

- "store_false" [relevant: dest]

Like "store_true", but stores a false value.

Example:

```python
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- "append" [relevant: type, dest, nargs, choices]

The option must be followed by an argument, which is appended to the list in dest. If no default value for dest is supplied, an empty list is automatically created when optparse first encounters this option on the command-line. If nargs > 1, multiple arguments are consumed, and a tuple of length nargs is appended to dest.

The defaults for type and dest are the same as for the "store" action.

Example:

```python
parser.add_option("-t", "--tracks", action="append", type="int")
```

If `-t3` is seen on the command-line, optparse does the equivalent of:

```python
options.tracks = []
options.tracks.append(int("3"))
```

If, a little later on, `--tracks=4` is seen, it does:
options.tracks.append(int("4"))

The `append` action calls the `append` method on the current value of the option. This means that any default value specified must have an `append` method. It also means that if the default value is non-empty, the default elements will be present in the parsed value for the option, with any values from the command line appended after those default values:

```python
>>> parser.add_option("--files", action="append", default=[
    >>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
    >>> opts.files
    ['~/.mypkg/defaults', 'overrides.mypkg']
```

- "append_const" [required: `const`; relevant: `dest`]

  Like "store_const", but the value `const` is appended to `dest`; as with "append", `dest` defaults to `None`, and an empty list is automatically created the first time the option is encountered.

- "count" [relevant: `dest`]

  Increment the integer stored at `dest`. If no default value is supplied, `dest` is set to zero before being incremented the first time.

  Example:

  ```python
  parser.add_option("-v", action="count", dest="verbosity")
  ```

  The first time `-v` is seen on the command line, `optparse` does the equivalent of:

  ```python
  options.verbosity = 0
  options.verbosity += 1
  ```
Every subsequent occurrence of `-v` results in

```
options.verbosity += 1
```

- **"callback"**  [required: `callback`; relevant: `type`, `nargs`, `callback_args`], `callback_kwargs`]

Call the function specified by `callback`, which is called as

```
func(option, opt_str, value, parser, *args, **kwargs)
```

See section **Option Callbacks** for more detail.

- **"help"**

Prints a complete help message for all the options in the current option parser. The help message is constructed from the `usage` string passed to OptionParser's constructor and the `help` string passed to every option.

If no `help` string is supplied for an option, it will still be listed in the help message. To omit an option entirely, use the special value `optparse.SUPPRESS_HELP`.

`optparse` automatically adds a `help` option to all OptionParsers, so you do not normally need to create one.

Example:

```
from optparse import OptionParser, SUPPRESS_HELP

# usually, a help option is added automatically, but that can be suppressed using the add_help_option argument
parser = OptionParser(add_help_option=False)

parser.add_option("-h", "--help", action="help")
parser.add_option("-v", action="store_true", dest="verbose"
```
If `optparse` sees either `-h` or `--help` on the command line, it will print something like the following help message to stdout (assuming `sys.argv[0]` is "foo.py"):

```
Usage: foo.py [options]
Options:
  -h, --help          Show this help message and exit
  -v                  Be moderately verbose
  --file=FILENAME     Input file to read data from
```

After printing the help message, `optparse` terminates your process with `sys.exit(0)`.

- "version"

Prints the version number supplied to the OptionParser to stdout and exits. The version number is actually formatted and printed by the `print_version()` method of OptionParser. Generally only relevant if the `version` argument is supplied to the OptionParser constructor. As with `help` options, you will rarely create `version` options, since `optparse` automatically adds them when needed.

15.5.3.6. Standard option types

`optparse` has six built-in option types: "string", "int", "long", "choice", "float" and "complex". If you need to add new option types, see section Extending `optparse`.

Arguments to string options are not checked or converted in any way: the text on the command line is stored in the destination (or
passed to the callback) as-is.

Integer arguments (type "int" or "long") are parsed as follows:

- if the number starts with 0x, it is parsed as a hexadecimal number
- if the number starts with 0, it is parsed as an octal number
- if the number starts with 0b, it is parsed as a binary number
- otherwise, the number is parsed as a decimal number

The conversion is done by calling either \texttt{int() or long()} with the appropriate base (2, 8, 10, or 16). If this fails, so will \texttt{optparse}, although with a more useful error message.

"float" and "complex" option arguments are converted directly with \texttt{float()} and \texttt{complex()}, with similar error-handling.

"choice" options are a subtype of "string" options. The \texttt{choices} option attribute (a sequence of strings) defines the set of allowed option arguments. \texttt{optparse.check_choice()} compares user-supplied option arguments against this master list and raises \texttt{OptionValueError} if an invalid string is given.

15.5.3.7. Parsing arguments

The whole point of creating and populating an OptionParser is to call its \texttt{parse_args()} method:

\begin{verbatim}
(options, args) = parser.parse_args(args=None, values=None)
\end{verbatim}

where the input parameters are

\begin{verbatim}
args
\end{verbatim}

the list of arguments to process (default: \texttt{sys.argv[1:]})
values

an optparse.Values object to store option arguments in (default: a new instance of values) – if you give an existing object, the option defaults will not be initialized on it

and the return values are

options

the same object that was passed in as values, or the optparse.Values instance created by optparse

args

the leftover positional arguments after all options have been processed

The most common usage is to supply neither keyword argument. If you supply values, it will be modified with repeated setattr() calls (roughly one for every option argument stored to an option destination) and returned by parse_args().

If parse_args() encounters any errors in the argument list, it calls the OptionParser’s error() method with an appropriate end-user error message. This ultimately terminates your process with an exit status of 2 (the traditional Unix exit status for command-line errors).

15.5.3.8. Querying and manipulating your option parser

The default behavior of the option parser can be customized slightly, and you can also poke around your option parser and see what’s there. OptionParser provides several methods to help you out:

OptionParser.disable_interspersed_args()

Set parsing to stop on the first non-option. For example, if -a and -b are both simple options that take no arguments, optparse
normally accepts this syntax:

```
prog -a arg1 -b arg2
```

and treats it as equivalent to

```
prog -a -b arg1 arg2
```

To disable this feature, call `disable_interspersed_args()`. This restores traditional Unix syntax, where option parsing stops with the first non-option argument.

Use this if you have a command processor which runs another command which has options of its own and you want to make sure these options don't get confused. For example, each command might have a different set of options.

OptionParser. **enable_interspersed_args()**

Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior.

OptionParser. **get_option(opt_str)**

Returns the Option instance with the option string `opt_str`, or `None` if no options have that option string.

OptionParser. **has_option(opt_str)**

Return true if the OptionParser has an option with option string `opt_str` (e.g., `-q` or `--verbose`).

OptionParser. **remove_option(opt_str)**

If the OptionParser has an option corresponding to `opt_str`, that option is removed. If that option provided any other option strings, all of those option strings become invalid. If `opt_str` does not occur in any option belonging to this OptionParser, raises
15.5.3.9. Conflicts between options

If you're not careful, it's easy to define options with conflicting option strings:

```python
parser.add_option("-n", "--dry-run", ...)  
...  
parser.add_option("-n", "--noisy", ...)
```

(This is particularly true if you've defined your own OptionParser subclass with some standard options.)

Every time you add an option, `optparse` checks for conflicts with existing options. If it finds any, it invokes the current conflict-handling mechanism. You can set the conflict-handling mechanism either in the constructor:

```python
parser = OptionParser(..., conflict_handler=handler)
```

or with a separate call:

```python
parser.set_conflict_handler(handler)
```

The available conflict handlers are:

- "error" (default)
  assume option conflicts are a programming error and raise `OptionConflictError`

- "resolve"
  resolve option conflicts intelligently (see below)

As an example, let's define an `OptionParser` that resolves conflicts intelligently and add conflicting options to it:
At this point, `optparse` detects that a previously-added option is already using the `-n` option string. Since `conflict_handler` is "resolve", it resolves the situation by removing `-n` from the earlier option’s list of option strings. Now `--dry-run` is the only way for the user to activate that option. If the user asks for help, the help message will reflect that:

```plaintext
Options:
- --dry-run do no harm
... 
- -n, --noisy be noisy
```

It’s possible to whittle away the option strings for a previously-added option until there are none left, and the user has no way of invoking that option from the command-line. In that case, `optparse` removes that option completely, so it doesn’t show up in help text or anywhere else. Carrying on with our existing OptionParser:

```python
parser.add_option("--dry-run", ..., help="new dry-run option")
```

At this point, the original `-n/--dry-run` option is no longer accessible, so `optparse` removes it, leaving this help text:

```plaintext
Options:
... 
- -n, --noisy be noisy
  --dry-run new dry-run option
```

### 15.5.3.10. Cleanup

OptionParser instances have several cyclic references. This should not be a problem for Python’s garbage collector, but you may wish to
break the cyclic references explicitly by calling `destroy()` on your OptionParser once you are done with it. This is particularly useful in long-running applications where large object graphs are reachable from your OptionParser.

### 15.5.3.11. Other methods

OptionParser supports several other public methods:

**OptionParser. `set_usage(usage)`**  
Set the usage string according to the rules described above for the `usage` constructor keyword argument. Passing `None` sets the default usage string; use `optparse.SUPPRESS_USAGE` to suppress a usage message.

**OptionParser. `print_usage(file=None)`**  
Print the usage message for the current program (`self.usage`) to `file` (default stdout). Any occurrence of the string `%prog` in `self.usage` is replaced with the name of the current program. Does nothing if `self.usage` is empty or not defined.

**OptionParser. `get_usage()`**  
Same as `print_usage()` but returns the usage string instead of printing it.

**OptionParser. `set_defaults(dest=value, ...)`**  
Set default values for several option destinations at once. Using `set_defaults()` is the preferred way to set default values for options, since multiple options can share the same destination. For example, if several “mode” options all set the same destination, any one of them can set the default, and the last one wins:

```python
parser.add_option("--advanced", action="store_const",
```
dest="mode", const="advanced", default="novice")  # overridden below
parser.add_option("--novice", action="store_const",
dest="mode", const="novice",
default="advanced")  # overrides above set

To avoid this confusion, use set_defaults():

parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
dest="mode", const="novice")
15.5.4. Option Callbacks

When `optparse`'s built-in actions and types aren't quite enough for your needs, you have two choices: extend `optparse` or define a callback option. Extending `optparse` is more general, but overkill for a lot of simple cases. Quite often a simple callback is all you need.

There are two steps to defining a callback option:

- define the option itself using the "callback" action
- write the callback; this is a function (or method) that takes at least four arguments, as described below

15.5.4.1. Defining a callback option

As always, the easiest way to define a callback option is by using the `OptionParser.add_option()` method. Apart from `action`, the only option attribute you must specify is `callback`, the function to call:

```python
parser.add_option("-c", action="callback", callback=my_callback)
```

`callback` is a function (or other callable object), so you must have already defined `my_callback()` when you create this callback option. In this simple case, `optparse` doesn't even know if `-c` takes any arguments, which usually means that the option takes no arguments—the mere presence of `-c` on the command-line is all it needs to know. In some circumstances, though, you might want your callback to consume an arbitrary number of command-line arguments. This is where writing callbacks gets tricky; it's covered later in this section.

`optparse` always passes four particular arguments to your callback, and it will only pass additional arguments if you specify them via
callback_args and callback_kwargs. Thus, the minimal callback function signature is:

```python
def my_callback(option, opt, value, parser):
```

The four arguments to a callback are described below.

There are several other option attributes that you can supply when you define a callback option:

**type**

has its usual meaning: as with the "store" or "append" actions, it instructs optparse to consume one argument and convert it to type. Rather than storing the converted value(s) anywhere, though, optparse passes it to your callback function.

**nargs**

also has its usual meaning: if it is supplied and > 1, optparse will consume nargs arguments, each of which must be convertible to type. It then passes a tuple of converted values to your callback.

**callback_args**

a tuple of extra positional arguments to pass to the callback

**callback_kwargs**

da dictionary of extra keyword arguments to pass to the callback

### 15.5.4.2. How callbacks are called

All callbacks are called as follows:

```python
func(option, opt_str, value, parser, *args, **kwargs)
```

where

**option**

is the Option instance that's calling the callback
opt_str

is the option string seen on the command-line that's triggering the callback. (If an abbreviated long option was used, opt_str will be the full, canonical option string—e.g. if the user puts --foo on the command-line as an abbreviation for --foobar, then opt_str will be "--foobar".)

value

is the argument to this option seen on the command-line. optparse will only expect an argument if type is set; the type of value will be the type implied by the option's type. If type for this option is None (no argument expected), then value will be None. If nargs > 1, value will be a tuple of values of the appropriate type.

parser

is the OptionParser instance driving the whole thing, mainly useful because you can access some other interesting data through its instance attributes:

parser.largs

the current list of leftover arguments, ie. arguments that have been consumed but are neither options nor option arguments. Feel free to modify parser.largs, e.g. by adding more arguments to it. (This list will become args, the second return value of parse_args().)

parser.rargs

the current list of remaining arguments, ie. with opt_str and value (if applicable) removed, and only the arguments following them still there. Feel free to modify parser.rargs, e.g. by consuming more arguments.

parser.values

the object where option values are by default stored (an instance of optparse.OptionValues). This lets callbacks use
the same mechanism as the rest of optparse for storing option values; you don't need to mess around with globals or closures. You can also access or modify the value(s) of any options already encountered on the command-line.

args
is a tuple of arbitrary positional arguments supplied via the callback_args option attribute.

kwargs
is a dictionary of arbitrary keyword arguments supplied via callback_kwargs.

15.5.4.3. Raising errors in a callback

The callback function should raise OptionValueError if there are any problems with the option or its argument(s). optparse catches this and terminates the program, printing the error message you supply to stderr. Your message should be clear, concise, accurate, and mention the option at fault. Otherwise, the user will have a hard time figuring out what he did wrong.

15.5.4.4. Callback example 1: trivial callback

Here's an example of a callback option that takes no arguments, and simply records that the option was seen:

```python
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True
parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

Of course, you could do that with the "store_true" action.

15.5.4.5. Callback example 2: check option order
Here's a slightly more interesting example: record the fact that \(-a\) is seen, but blow up if it comes after \(-b\) in the command-line.

```python
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback", callback=check_order
parser.add_option("-b", action="store_true", dest="b")
```

15.5.4.6. Callback example 3: check option order (generalized)

If you want to re-use this callback for several similar options (set a flag, but blow up if \(-b\) has already been seen), it needs a bit of work: the error message and the flag that it sets must be generalized.

```python
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_st
setattr(parser.values, option.dest, 1)
...
parser.add_option("-a", action="callback", callback=check_order
parser.add_option("-b", action="store_true", dest="b")
parsing.add_option("-c", action="callback", callback=check_order
```

15.5.4.7. Callback example 4: check arbitrary condition

Of course, you could put any condition in there—you're not limited to checking the values of already-defined options. For example, if you have options that should not be called when the moon is full, all you have to do is this:

```python
def check_moon(option, opt_str, value, parser):
```
if is_moon_full():
    raise OptionValueError("%s option invalid when moon is % opt_str")
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
    action="callback", callback=check_moon, dest=

(The definition of is_moon_full() is left as an exercise for the reader.)

15.5.4.8. Callback example 5: fixed arguments

Things get slightly more interesting when you define callback options that take a fixed number of arguments. Specifying that a callback option takes arguments is similar to defining a "store" or "append" option: if you define type, then the option takes one argument that must be convertible to that type; if you further define nargs, then the option takes nargs arguments.

Here's an example that just emulates the standard "store" action:

def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
    action="callback", callback=store_value,
    type="int", nargs=3, dest="foo")

Note that optparse takes care of consuming 3 arguments and converting them to integers for you; all you have to do is store them. (Or whatever; obviously you don't need a callback for this example.)

15.5.4.9. Callback example 6: variable arguments

Things get hairy when you want an option to take a variable number
of arguments. For this case, you must write a callback, as `optparse` doesn't provide any built-in capabilities for it. And you have to deal with certain intricacies of conventional Unix command-line parsing that `optparse` normally handles for you. In particular, callbacks should implement the conventional rules for bare `-` and `--` arguments:

- either `--` or `-` can be option arguments
- bare `--` (if not the argument to some option): halt command-line processing and discard the `--`
- bare `-` (if not the argument to some option): halt command-line processing but keep the `-` (append it to `parser.largs`)

If you want an option that takes a variable number of arguments, there are several subtle, tricky issues to worry about. The exact implementation you choose will be based on which trade-offs you're willing to make for your application (which is why `optparse` doesn't support this sort of thing directly).

Nevertheless, here's a stab at a callback for an option with variable arguments:

```python
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []

def floatable(str):
    try:
        float(str)
        return True
    except ValueError:
        return False

for arg in parser.rargs:
    # stop on --foo like options
    if arg[:2] == "--" and len(arg) > 2:
        break
    # stop on -a, but not on -3 or -3.0
```
if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
    break
    value.append(arg)

delete parser.rargs[:len(value)]
    setattr(parser.values, option.dest, value)

...parser.add_option("-c", "--callback", dest="vararg_attr",
    action="callback", callback=vararg_callback)
15.5.5. Extending optparse

Since the two major controlling factors in how optparse interprets command-line options are the action and type of each option, the most likely direction of extension is to add new actions and new types.

15.5.5.1. Adding new types

To add new types, you need to define your own subclass of optparse's Option class. This class has a couple of attributes that define optparse's types: TYPES and TYPE_CHECKER.

Option. TYPES
A tuple of type names; in your subclass, simply define a new tuple TYPES that builds on the standard one.

Option. TYPE_CHECKER
A dictionary mapping type names to type-checking functions. A type-checking function has the following signature:

```python
def check_mytype(option, opt, value)
```

where option is an Option instance, opt is an option string (e.g., -f), and value is the string from the command line that must be checked and converted to your desired type. check_mytype() should return an object of the hypothetical type mytype. The value returned by a type-checking function will wind up in the OptionValues instance returned by OptionParser.parse_args(), or be passed to a callback as the value parameter.

Your type-checking function should raise OptionValueError if it encounters any problems. OptionValueError takes a single string
argument, which is passed as-is to OptionParser's error() method, which in turn prepends the program name and the string "error:" and prints everything to stderr before terminating the process.

Here's a silly example that demonstrates adding a complex option type to parse Python-style complex numbers on the command line. (This is even sillier than it used to be, because optparse 1.3 added built-in support for complex numbers, but never mind.)

First, the necessary imports:

```python
from copy import copy
from optparse import Option, OptionValueError
```

You need to define your type-checker first, since it's referred to later (in the TYPE_CHECKER class attribute of your Option subclass):

```python
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError("option %s: invalid complex value: %r" % (opt, value))
```

Finally, the Option subclass:

```python
class MyOption(Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(If we didn't make a copy() of Option.TYPE_CHECKER, we would end up modifying the TYPE_CHECKER attribute of optparse's Option class. This being Python, nothing stops you from doing that except good manners and common sense.)
That’s it! Now you can write a script that uses the new option type just like any other optparse-based script, except you have to instruct your OptionParser to use MyOption instead of Option:

```python
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

Alternately, you can build your own option list and pass it to OptionParser; if you don’t use `add_option()` in the above way, you don’t need to tell OptionParser which option class to use:

```python
option_list = [MyOption("-c", action="store", type="complex", d
parser = OptionParser(option_list=option_list)
```

### 15.5.5.2. Adding new actions

Adding new actions is a bit trickier, because you have to understand that optparse has a couple of classifications for actions:

“store” actions
actions that result in optparse storing a value to an attribute of the current OptionValues instance; these options require a `dest` attribute to be supplied to the Option constructor.

“typed” actions
actions that take a value from the command line and expect it to be of a certain type; or rather, a string that can be converted to a certain type. These options require a `type` attribute to the Option constructor.

These are overlapping sets: some default “store” actions are "store", "store_const", "append", and "count", while the default “typed” actions are "store", "append", and "callback".

When you add an action, you need to categorize it by listing it in at
least one of the following class attributes of Option (all are lists of strings):

Option. **ACTIONS**
   All actions must be listed in ACTIONS.

Option. **STORE_ACTIONS**
   “store” actions are additionally listed here.

Option. **TYPED_ACTIONS**
   “typed” actions are additionally listed here.

Option. **ALWAYS_TYPED_ACTIONS**
   Actions that always take a type (i.e. whose options always take a value) are additionally listed here. The only effect of this is that `optparse` assigns the default type, "string", to options with no explicit type whose action is listed in **ALWAYS_TYPED_ACTIONS**.

In order to actually implement your new action, you must override Option’s `take_action()` method and add a case that recognizes your action.

For example, let’s add an "extend" action. This is similar to the standard "append" action, but instead of taking a single value from the command-line and appending it to an existing list, "extend" will take multiple values in a single comma-delimited string, and extend an existing list with them. That is, if `--names` is an "extend" option of type "string", the command line

```
--names=foo,bar --names blah --names ding,dong
```

would result in a list

```
["foo", "bar", "blah", "ding", "dong"]
```
Again we define a subclass of Option:

```python
class MyOption(Option):
    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(self, action, dest, opt, value, values, parser)
```

Features of note:

- "extend" both expects a value on the command-line and stores that value somewhere, so it goes in both `STORE_ACTIONS` and `TYPED_ACTIONS`.

- to ensure that `optparse` assigns the default type of "string" to "extend" actions, we put the "extend" action in `ALWAYS_TYPED_ACTIONS` as well.

- `MyOption.take_action()` implements just this one new action, and passes control back to `Option.take_action()` for the standard `optparse` actions.

- `values` is an instance of the `optparse.parser.Values` class, which provides the very useful `ensure_value()` method. `ensure_value()` is essentially `getattr()` with a safety valve; it is called as

```python
values.ensure_value(attr, value)
```
If the `attr` attribute of `values` doesn’t exist or is `None`, then `ensure_value()` first sets it to `value`, and then returns `value`. This is very handy for actions like `"extend"`, `"append"`, and `"count"`, all of which accumulate data in a variable and expect that variable to be of a certain type (a list for the first two, an integer for the latter). Using `ensure_value()` means that scripts using your action don’t have to worry about setting a default value for the option destinations in question; they can just leave the default as `None` and `ensure_value()` will take care of getting it right when it’s needed.
15.6. getopt — C-style parser for command line options

Source code: Lib/getopt.py

Note: The getopt module is a parser for command line options whose API is designed to be familiar to users of the C getopt() function. Users who are unfamiliar with the C getopt() function or who would like to write less code and get better help and error messages should consider using the argparse module instead.

This module helps scripts to parse the command line arguments in sys.argv. It supports the same conventions as the Unix getopt() function (including the special meanings of arguments of the form ‘-‘ and ‘--‘). Long options similar to those supported by GNU software may be used as well via an optional third argument.

This module provides two functions and an exception:

getopt.getopt(args, options[, long_options])

Parses command line options and parameter list. args is the argument list to be parsed, without the leading reference to the running program. Typically, this means sys.argv[1:]. options is the string of option letters that the script wants to recognize, with options that require an argument followed by a colon (':'; i.e., the same format that Unix getopt() uses).

Note: Unlike GNU getopt(), after a non-option argument, all further arguments are considered also non-options. This is similar to the way non-GNU Unix systems work.
long_options, if specified, must be a list of strings with the names of the long options which should be supported. The leading '--' characters should not be included in the option name. Long options which require an argument should be followed by an equal sign ('='). Optional arguments are not supported. To accept only long options, options should be an empty string. Long options on the command line can be recognized so long as they provide a prefix of the option name that matches exactly one of the accepted options. For example, if long_options is ['foo', 'frob'], the option --fo will match as --foo, but --f will not match uniquely, so GetoptError will be raised.

The return value consists of two elements: the first is a list of (option, value) pairs; the second is the list of program arguments left after the option list was stripped (this is a trailing slice of args). Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option'), and the option argument as its second element, or an empty string if the option has no argument. The options occur in the list in the same order in which they were found, thus allowing multiple occurrences. Long and short options may be mixed.

gnu_getopt (args, options[, long_options])

This function works like getopt(), except that GNU style scanning mode is used by default. This means that option and non-option arguments may be intermixed. The getopt() function stops processing options as soon as a non-option argument is encountered.

If the first character of the option string is '+', or if the environment variable POSIXLY_CORRECT is set, then option processing stops as soon as a non-option argument is
exception getopt.GetoptError

This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none. The argument to the exception is a string indicating the cause of the error. For long options, an argument given to an option which does not require one will also cause this exception to be raised. The attributes msg and opt give the error message and related option; if there is no specific option to which the exception relates, opt is an empty string.

Changed in version 1.6: Introduced GetoptError as a synonym for error.

exception getopt.error

Alias for GetoptError; for backward compatibility.

An example using only Unix style options:

```python
g>>> import getopt
g>>> args = '-a -b -cfoo -d bar a1 a2'.split()
g>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
g>>> optlist, args = getopt.getopt(args, 'abc:d:')[0]
g>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]  
g>>> args
['a1', 'a2']
```

Using long option names is equally easy:

```python
g>>> s = '--condition=foo --testing --output-file abc.def -x a1
g>>> args = s.split()
g>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x']
```
In a script, typical usage is something like this:

```python
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "usage()"],
                                 ["output", "--output"])
    except getopt.GetoptError as err:
        # print help information and exit:
        print str(err)  # will print something like "option -a not recognized"
        sys.exit(2)
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()  # will print something like "option -a not recognized"
        elif o in ("-o", "--output"):
            output = a
        else:
            assert False, "unhandled option"
    if __name__ == "__main__":
        main()
```

Note that an equivalent command line interface could be produced with less code and more informative help and error messages by using the `argparse` module:

```python
import argparse
```
```python
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    args = parser.parse_args()
    # ... do something with args.output ...  
    # ... do something with args.verbose ..
```

See also:

**Module argparse**

Alternative command line option and argument parsing library.
15.7. `logging` — Logging facility for Python

Source
Lib/logging/__init__.py

---

New in version 2.3.

This module defines functions and classes which implement a flexible event logging system for applications and libraries.

The key benefit of having the logging API provided by a standard library module is that all Python modules can participate in logging, so your application log can include your own messages integrated with messages from third-party modules.

The module provides a lot of functionality and flexibility. If you are unfamiliar with logging, the best way to get to grips with it is to see the tutorials (see the links on the right).

The basic classes defined by the module, together with their functions, are listed below.

- Loggers expose the interface that application code directly uses.
- Handlers send the log records (created by loggers) to the appropriate destination.
- Filters provide a finer grained facility for determining which log records to output.
- Formatters specify the layout of log records in the final output.
15.7.1. Logger Objects

Loggers have the following attributes and methods. Note that Loggers are never instantiated directly, but always through the module-level function `logging.getLogger(name)`. Multiple calls to `getLogger()` with the same name will always return a reference to the same Logger object.

The `name` is potentially a period-separated hierarchical value, like `foo.bar.baz` (though it could also be just plain `foo`, for example). Loggers that are further down in the hierarchical list are children of loggers higher up in the list. For example, given a logger with a name of `foo`, loggers with names of `foo.bar`, `foo.bar.baz`, and `foo.bam` are all descendants of `foo`. The logger name hierarchy is analogous to the Python package hierarchy, and identical to it if you organise your loggers on a per-module basis using the recommended construction `logging.getLogger(__name__)`. That's because in a module, `__name__` is the module's name in the Python package namespace.

```python
class logging.Logger

Logger.propagate
```

If this evaluates to true, events logged to this logger will be passed to the handlers of higher level (ancestor) loggers, in addition to any handlers attached to this logger. Messages are passed directly to the ancestor loggers' handlers - neither the level nor filters of the ancestor loggers in question are considered.

If this evaluates to false, logging messages are not passed to the handlers of ancestor loggers.
The constructor sets this attribute to True.

**Note:** If you attach a handler to a logger and one or more of its ancestors, it may emit the same record multiple times. In general, you should not need to attach a handler to more than one logger - if you just attach it to the appropriate logger which is highest in the logger hierarchy, then it will see all events logged by all descendant loggers, provided that their propagate setting is left set to True. A common scenario is to attach handlers only to the root logger, and to let propagation take care of the rest.

**Logger. `setLevel(lvl)`**

Sets the threshold for this logger to `lvl`. Logging messages which are less severe than `lvl` will be ignored. When a logger is created, the level is set to **`NOTSET`** (which causes all messages to be processed when the logger is the root logger, or delegation to the parent when the logger is a non-root logger). Note that the root logger is created with level **`WARNING`**.

The term 'delegation to the parent' means that if a logger has a level of NOTSET, its chain of ancestor loggers is traversed until either an ancestor with a level other than NOTSET is found, or the root is reached.

If an ancestor is found with a level other than NOTSET, then that ancestor’s level is treated as the effective level of the logger where the ancestor search began, and is used to determine how a logging event is handled.

If the root is reached, and it has a level of NOTSET, then all messages will be processed. Otherwise, the root's level will be used as the effective level.
See **Logging Levels** for a list of levels.

**Logger. isEnabledFor**(lvl)**

Indicates if a message of severity *lvl* would be processed by this logger. This method checks first the module-level level set by `logging.disable(lvl)` and then the logger's effective level as determined by `getEffectiveLevel()`.

**Logger. getEffectiveLevel()**

Indicates the effective level for this logger. If a value other than `NOTSET` has been set using `setLevel()`, it is returned. Otherwise, the hierarchy is traversed towards the root until a value other than `NOTSET` is found, and that value is returned. The value returned is an integer, typically one of `logging.DEBUG`, `logging.INFO` etc.

**Logger. getChild**(suffix)**

Returns a logger which is a descendant to this logger, as determined by the suffix. Thus, `logging.getLogger('abc').getChild('def.ghi')` would return the same logger as would be returned by `logging.getLogger('abc.def.ghi')`. This is a convenience method, useful when the parent logger is named using e.g. `__name__` rather than a literal string.

*New in version 2.7.*

**Logger. debug**(msg, *args, **kwargs)**

Logs a message with level `DEBUG` on this logger. The *msg* is the message format string, and the *args* are the arguments which are merged into *msg* using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)
There are two keyword arguments in `kwargs` which are inspected: `exc_info` which, if it does not evaluate as false, causes exception information to be added to the logging message. If an exception tuple (in the format returned by `sys.exc_info()`) is provided, it is used; otherwise, `sys.exc_info()` is called to get the exception information.

The second keyword argument is `extra` which can be used to pass a dictionary which is used to populate the `__dict__` of the LogRecord created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

```python
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset', e, extra=d)
```

would print something like

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs Protocol problem: connection reset
```

The keys in the dictionary passed in `extra` should not clash with the keys used by the logging system. (See the `Formatter` documentation for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the `Formatter` has been set up with a format string which expects ‘clientip’ and ‘user’ in the attribute dictionary of the LogRecord. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case, you always
need to pass the extra dictionary with these keys.

While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-threaded servers where the same code executes in many contexts, and interesting conditions which arise are dependent on this context (such as remote client IP address and authenticated user name, in the above example). In such circumstances, it is likely that specialized Formatters would be used with particular Handlers.

Logger. **info**(msg, *args, **kwargs)
Logs a message with level INFO on this logger. The arguments are interpreted as for debug().

Logger. **warning**(msg, *args, **kwargs)
Logs a message with level WARNING on this logger. The arguments are interpreted as for debug().

Logger. **error**(msg, *args, **kwargs)
Logs a message with level ERROR on this logger. The arguments are interpreted as for debug().

Logger. **critical**(msg, *args, **kwargs)
Logs a message with level CRITICAL on this logger. The arguments are interpreted as for debug().

Logger. **log**(lvl, msg, *args, **kwargs)
Logs a message with integer level lvl on this logger. The other arguments are interpreted as for debug().

Logger. **exception**(msg, *args, **kwargs)
Logs a message with level ERROR on this logger. The arguments are interpreted as for debug(), except that any passed exc_info is not inspected. Exception info is always added to the logging
message. This method should only be called from an exception handler.

Logger.addFilter(filt)
    Adds the specified filter filt to this logger.

Logger.removeFilter(filt)
    Removes the specified filter filt from this logger.

Logger.filter(record)
    Applies this logger’s filters to the record and returns a true value if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be processed (passed to handlers). If one returns a false value, no further processing of the record occurs.

Logger.addHandler(hdlr)
    Adds the specified handler hdlr to this logger.

Logger.removeHandler(hdlr)
    Removes the specified handler hdlr from this logger.

Logger.findCaller()
    Finds the caller’s source filename and line number. Returns the filename, line number and function name as a 3-element tuple.

    Changed in version 2.4: The function name was added. In earlier versions, the filename and line number were returned as a 2-element tuple.

Logger.handle(record)
    Handles a record by passing it to all handlers associated with this logger and its ancestors (until a false value of propagate is found). This method is used for unpickled records received from a socket, as well as those created locally. Logger-level filtering is
applied using `filter()`.

Logger's `makeRecord(name, lvl, fn, lno, msg, args, exc_info, func=None, extra=None)`

This is a factory method which can be overridden in subclasses to create specialized `LogRecord` instances.

*Changed in version 2.5: `func` and `extra` were added.*
15.7.2. Logging Levels

The numeric values of logging levels are given in the following table. These are primarily of interest if you want to define your own levels, and need them to have specific values relative to the predefined levels. If you define a level with the same numeric value, it overwrites the predefined value; the predefined name is lost.

<table>
<thead>
<tr>
<th>Level</th>
<th>Numeric value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRITICAL</td>
<td>50</td>
</tr>
<tr>
<td>ERROR</td>
<td>40</td>
</tr>
<tr>
<td>WARNING</td>
<td>30</td>
</tr>
<tr>
<td>INFO</td>
<td>20</td>
</tr>
<tr>
<td>DEBUG</td>
<td>10</td>
</tr>
<tr>
<td>NOTSET</td>
<td>0</td>
</tr>
</tbody>
</table>
15.7.3. Handler Objects

Handlers have the following attributes and methods. Note that `Handler` is never instantiated directly; this class acts as a base for more useful subclasses. However, the `__init__()` method in subclasses needs to call `Handler.__init__()`.

```
Handler.__init__(level=NOTSET)
```

Initializes the `Handler` instance by setting its level, setting the list of filters to the empty list and creating a lock (using `createLock()`) for serializing access to an I/O mechanism.

```
Handler.createLock()
```

Initializes a thread lock which can be used to serialize access to underlying I/O functionality which may not be threadsafe.

```
Handler.acquire()
```

Acquires the thread lock created with `createLock()`.

```
Handler.release()
```

Releases the thread lock acquired with `acquire()`.

```
Handler.setLevel(lvl)
```

Sets the threshold for this handler to `lvl`. Logging messages which are less severe than `lvl` will be ignored. When a handler is created, the level is set to `NOTSET` (which causes all messages to be processed).

See Logging Levels for a list of levels.

```
Handler.setFormatter(form)
```

Sets the `Formatter` for this handler to `form`. 


**Handler`.addFilter(filt)`**

Adds the specified filter `filt` to this handler.

**Handler`.removeFilter(filt)`**

Removes the specified filter `filt` from this handler.

**Handler`.filter(record)`**

Applies this handler’s filters to the record and returns a true value if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be emitted. If one returns a false value, the handler will not emit the record.

**Handler`.flush()`**

Ensure all logging output has been flushed. This version does nothing and is intended to be implemented by subclasses.

**Handler`.close()`**

Tidy up any resources used by the handler. This version does no output but removes the handler from an internal list of handlers which is closed when `shutdown()` is called. Subclasses should ensure that this gets called from overridden `close()` methods.

**Handler`.handle(record)`**

Conditionally emits the specified logging record, depending on filters which may have been added to the handler. Wraps the actual emission of the record with acquisition/release of the I/O thread lock.

**Handler`.handleError(record)`**

This method should be called from handlers when an exception is encountered during an `emit()` call. If the module-level attribute `raiseExceptions` is `False`, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested
in application errors. You could, however, replace this with a
custom handler if you wish. The specified record is the one which
was being processed when the exception occurred. (The default
value of raiseExceptions is True, as that is more useful during
development).

Handler. **format**(record)
Do formatting for a record - if a formatter is set, use it. Otherwise,
use the default formatter for the module.

Handler. **emit**(record)
Do whatever it takes to actually log the specified logging record.
This version is intended to be implemented by subclasses and so
raises a NotImplementedError.

For a list of handlers included as standard, see `logging.handlers`. 
15.7.4. Formatter Objects

Formatter objects have the following attributes and methods. They are responsible for converting a LogRecord to (usually) a string which can be interpreted by either a human or an external system. The base Formatter allows a formatting string to be specified. If none is supplied, the default value of '%%(message)s' is used, which just includes the message in the logging call. To have additional items of information in the formatted output (such as a timestamp), keep reading.

A Formatter can be initialized with a format string which makes use of knowledge of the LogRecord attributes - such as the default value mentioned above making use of the fact that the user's message and arguments are pre-formatted into a LogRecord's message attribute. This format string contains standard Python %-style mapping keys. See section String Formatting Operations for more information on string formatting.

The useful mapping keys in a LogRecord are given in the section on LogRecord attributes.

class logging.Formatter(fmt=None, datefmt=None)

Returns a new instance of the Formatter class. The instance is initialized with a format string for the message as a whole, as well as a format string for the date/time portion of a message. If no fmt is specified, '%%(message)s' is used. If no datefmt is specified, the ISO8601 date format is used.

format(record)

The record's attribute dictionary is used as the operand to a string formatting operation. Returns the resulting string. Before formatting the dictionary, a couple of preparatory steps
are carried out. The message attribute of the record is computed using \textit{msg} \% \textit{args}. If the formatting string contains \\
'(asctime)', \texttt{formatTime()} is called to format the event time. If there is exception information, it is formatted using \texttt{formatException()} and appended to the message. Note that the formatted exception information is cached in attribute \textit{exc_text}. This is useful because the exception information can be pickled and sent across the wire, but you should be careful if you have more than one \texttt{Formatter} subclass which customizes the formatting of exception information. In this case, you will have to clear the cached value after a formatter has done its formatting, so that the next formatter to handle the event doesn’t use the cached value but recalculates it afresh.

\texttt{formatTime(record, datefmt=None)}

This method should be called from \texttt{format()} by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behavior is as follows: if \textit{datefmt} (a string) is specified, it is used with \texttt{time.strftime()} to format the creation time of the record. Otherwise, the ISO8601 format is used. The resulting string is returned.

This function uses a user-configurable function to convert the creation time to a tuple. By default, \texttt{time.localtime()} is used; to change this for a particular formatter instance, set the \texttt{converter} attribute to a function with the same signature as \texttt{time.localtime()} or \texttt{time.gmtime()}. To change it for all formatters, for example if you want all logging times to be shown in GMT, set the \texttt{converter} attribute in the \texttt{Formatter} class.

\texttt{formatException(exc_info)}
Formats the specified exception information (a standard exception tuple as returned by `sys.exc_info()`) as a string. This default implementation just uses `traceback.print_exception()`. The resulting string is returned.
15.7.5. Filter Objects

Filters can be used by Handlers and Loggers for more sophisticated filtering than is provided by levels. The base filter class only allows events which are below a certain point in the logger hierarchy. For example, a filter initialized with ‘A.B’ will allow events logged by loggers ‘A.B’, ‘A.B.C’, ‘A.B.C.D’, ‘A.B.D’ etc. but not ‘A.BB’, ‘B.A.B’ etc. If initialized with the empty string, all events are passed.

class logging.Filter(name='')

Returns an instance of the Filter class. If name is specified, it names a logger which, together with its children, will have its events allowed through the filter. If name is the empty string, allows every event.

filter(record)

Is the specified record to be logged? Returns zero for no, nonzero for yes. If deemed appropriate, the record may be modified in-place by this method.

Note that filters attached to handlers are consulted before an event is emitted by the handler, whereas filters attached to loggers are consulted whenever an event is logged (using debug(), info(), etc.), before sending an event to handlers. This means that events which have been generated by descendant loggers will not be filtered by a logger's filter setting, unless the filter has also been applied to those descendant loggers.

You don’t actually need to subclass Filter: you can pass any instance which has a filter method with the same semantics.

Although filters are used primarily to filter records based on more sophisticated criteria than levels, they get to see every record which
is processed by the handler or logger they're attached to: this can be useful if you want to do things like counting how many records were processed by a particular logger or handler, or adding, changing or removing attributes in the LogRecord being processed. Obviously changing the LogRecord needs to be done with some care, but it does allow the injection of contextual information into logs (see Using Filters to impart contextual information).
15.7.6. LogRecord Objects

*LogRecord* instances are created automatically by the *Logger* every time something is logged, and can be created manually via *makeLogRecord()* (for example, from a pickled event received over the wire).

```
class logging.LogRecord(name, level, pathname, lineno, msg, args, exc_info, func=None)

Contains all the information pertinent to the event being logged.
```

The primary information is passed in *msg* and *args*, which are combined using *msg % args* to create the *message* field of the record.

**Parameters:**

- **name** – The name of the logger used to log the event represented by this LogRecord. Note that this name will always have this value, even though it may be emitted by a handler attached to a different (ancestor) logger.
- **level** – The numeric level of the logging event (one of DEBUG, INFO etc.) Note that this is converted to two attributes of the LogRecord: `levelno` for the numeric value and `levelname` for the corresponding level name.
- **pathname** – The full pathname of the source file where the logging call was made.
- **lineno** – The line number in the source file where the logging call was made.
- **msg** – The event description message, possibly a format string with placeholders for variable data.
- **args** – Variable data to merge into the *msg*
argument to obtain the event description.

- **exc_info** – An exception tuple with the current exception information, or **None** if no exception information is available.
- **func** – The name of the function or method from which the logging call was invoked.

*Changed in version 2.5: func was added.*

**getMessage()**
Returns the message for this **LogRecord** instance after merging any user-supplied arguments with the message. If the user-supplied message argument to the logging call is not a string, **str()** is called on it to convert it to a string. This allows use of user-defined classes as messages, whose **__str__** method can return the actual format string to be used.
15.7.7. LogRecord attributes

The LogRecord has a number of attributes, most of which are derived from the parameters to the constructor. (Note that the names do not always correspond exactly between the LogRecord constructor parameters and the LogRecord attributes.) These attributes can be used to merge data from the record into the format string. The following table lists (in alphabetical order) the attribute names, their meanings and the corresponding placeholder in a %-style format string.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>args</td>
<td>You shouldn't need to format this yourself.</td>
<td>The tuple of arguments merged into msg to produce message, or a dict whose values are used for the merge (when there is only one argument, and it is a dictionary).</td>
</tr>
<tr>
<td>asctime</td>
<td>%(asctime)s</td>
<td>Human-readable time when the LogRecord was created. By default this is of the form ‘2003-07-08 16:49:45,896’ (the numbers after the comma are millisecond portion of the time).</td>
</tr>
<tr>
<td>created</td>
<td>%(created)f</td>
<td>Time when the LogRecord was created (as returned by time.time()).</td>
</tr>
<tr>
<td>exc_info</td>
<td>You shouldn't need to format this yourself.</td>
<td>Exception tuple (à la sys.exc_info) or, if no exception has occurred, None.</td>
</tr>
<tr>
<td>filename</td>
<td>%(filename)s</td>
<td>Filename portion of pathname.</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>funcName</td>
<td>Name of function containing the logging call.</td>
<td></td>
</tr>
<tr>
<td>levelname</td>
<td>Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL').</td>
<td></td>
</tr>
<tr>
<td>levelname</td>
<td>Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL).</td>
<td></td>
</tr>
<tr>
<td>lineno</td>
<td>Source line number where the logging call was issued (if available).</td>
<td></td>
</tr>
<tr>
<td>module</td>
<td>Module (name portion of filename).</td>
<td></td>
</tr>
<tr>
<td>msecs</td>
<td>Millisecond portion of the time when the LogRecord was created.</td>
<td></td>
</tr>
<tr>
<td>message</td>
<td>The logged message, computed as msg % args. This is set when Formatter.format() is invoked.</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>Name of the logger used to log the call.</td>
<td></td>
</tr>
<tr>
<td>pathname</td>
<td>Full pathname of the source file where the logging call was issued (if available).</td>
<td></td>
</tr>
<tr>
<td>process</td>
<td>Process ID (if available).</td>
<td></td>
</tr>
<tr>
<td>processName</td>
<td>Process name (if available).</td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Format</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>relativeCreated</td>
<td>%relativeCreatedd</td>
<td>Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded.</td>
</tr>
<tr>
<td>thread</td>
<td>%threadd</td>
<td>Thread ID (if available).</td>
</tr>
<tr>
<td>threadName</td>
<td>%threadNamel</td>
<td>Thread name (if available).</td>
</tr>
</tbody>
</table>

*Changed in version 2.5: funcName was added.*

*Changed in version 2.6: processName was added.*
15.7.8. LoggerAdapter Objects

LoggerAdapter instances are used to conveniently pass contextual information into logging calls. For a usage example, see the section on adding contextual information to your logging output.

New in version 2.6.

class logging.LoggerAdapter(logger, extra)
    Returns an instance of LoggerAdapter initialized with an underlying Logger instance and a dict-like object.

process(msg, kwargs)
    Modifies the message and/or keyword arguments passed to a logging call in order to insert contextual information. This implementation takes the object passed as extra to the constructor and adds it to kwargs using key ‘extra’. The return value is a (msg, kwargs) tuple which has the (possibly modified) versions of the arguments passed in.

In addition to the above, LoggerAdapter supports the following methods of Logger: debug(), info(), warning(), error(), exception(), critical(), log() and isEnabledFor(). These methods have the same signatures as their counterparts in Logger, so you can use the two types of instances interchangeably for these calls.

Changed in version 2.7: The isEnabledFor() method was added to LoggerAdapter. This method delegates to the underlying logger.
15.7.9. Thread Safety

The logging module is intended to be thread-safe without any special work needing to be done by its clients. It achieves this though using threading locks; there is one lock to serialize access to the module’s shared data, and each handler also creates a lock to serialize access to its underlying I/O.

If you are implementing asynchronous signal handlers using the `signal` module, you may not be able to use logging from within such handlers. This is because lock implementations in the `threading` module are not always re-entrant, and so cannot be invoked from such signal handlers.
15.7.10. Module-Level Functions

In addition to the classes described above, there are a number of module-level functions.

`logging.getLogger([name])`

Return a logger with the specified name or, if no name is specified, return a logger which is the root logger of the hierarchy. If specified, the name is typically a dot-separated hierarchical name like “a”, “a.b” or “a.b.c.d”. Choice of these names is entirely up to the developer who is using logging.

All calls to this function with a given name return the same logger instance. This means that logger instances never need to be passed between different parts of an application.

`logging.getLoggerClass()`

Return either the standard `Logger` class, or the last class passed to `setLoggerClass()`. This function may be called from within a new class definition, to ensure that installing a customized `Logger` class will not undo customizations already applied by other code. For example:

```python
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

`logging.debug(msg[, *args[, **kwargs]])`

Logs a message with level `DEBUG` on the root logger. The `msg` is the message format string, and the `args` are the arguments which are merged into `msg` using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)
There are two keyword arguments in `kwargs` which are inspected: `exc_info` which, if it does not evaluate as false, causes exception information to be added to the logging message. If an exception tuple (in the format returned by `sys.exc_info()`) is provided, it is used; otherwise, `sys.exc_info()` is called to get the exception information.

The other optional keyword argument is `extra` which can be used to pass a dictionary which is used to populate the `__dict__` of the LogRecord created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

```python
FORMAT = "%(asctime)-15s %(clientip)s %(user)-8s %(message)s"
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logging.warning("Protocol problem: %s", "connection reset", extra=d)
```

would print something like:

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection reset
```

The keys in the dictionary passed in `extra` should not clash with the keys used by the logging system. (See the `Formatter` documentation for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the `Formatter` has been set up with a format string which expects `clientip` and `user` in the attribute dictionary of the LogRecord. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case, you always need to pass the `extra` dictionary with these keys.
While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-threaded servers where the same code executes in many contexts, and interesting conditions which arise are dependent on this context (such as remote client IP address and authenticated user name, in the above example). In such circumstances, it is likely that specialized `Formatter`s would be used with particular `Handlers`.

*Changed in version 2.5: extra was added.*

```python
logging.info(msg[, args[, kwargs]])
```

Logs a message with level `INFO` on the root logger. The arguments are interpreted as for `debug()`.

```python
logging.warning(msg[, args[, kwargs]])
```

Logs a message with level `WARNING` on the root logger. The arguments are interpreted as for `debug()`.

```python
logging.error(msg[, args[, kwargs]])
```

Logs a message with level `ERROR` on the root logger. The arguments are interpreted as for `debug()`.

```python
logging.critical(msg[, args[, kwargs]])
```

Logs a message with level `CRITICAL` on the root logger. The arguments are interpreted as for `debug()`.

```python
logging.exception(msg[, args[, kwargs]])
```

Logs a message with level `ERROR` on the root logger. The arguments are interpreted as for `debug()`, except that any passed `exc_info` is not inspected. Exception info is always added to the logging message. This function should only be called from an exception handler.
logging.\texttt{log}(\texttt{level}, \texttt{msg}[, \texttt{*args}[, \texttt{**kwargs}]])

Logs a message with level \texttt{level} on the root logger. The other arguments are interpreted as for \texttt{debug()}.

\begin{quote}
\textbf{Note:} The above module-level convenience functions, which delegate to the root logger, call \texttt{basicConfig()} to ensure that at least one handler is available. Because of this, they should not be used in threads, in versions of Python earlier than 2.7.1 and 3.2, unless at least one handler has been added to the root logger before the threads are started. In earlier versions of Python, due to a thread safety shortcoming in \texttt{basicConfig()}, this can (under rare circumstances) lead to handlers being added multiple times to the root logger, which can in turn lead to multiple messages for the same event.
\end{quote}

logging.\texttt{disable}(\texttt{lvl})

Provides an overriding level \texttt{lvl} for all loggers which takes precedence over the logger’s own level. When the need arises to temporarily throttle logging output down across the whole application, this function can be useful. Its effect is to disable all logging calls of severity \texttt{lvl} and below, so that if you call it with a value of \texttt{INFO}, then all \texttt{INFO} and \texttt{DEBUG} events would be discarded, whereas those of severity \texttt{WARNING} and above would be processed according to the logger’s effective level. If \texttt{logging.disable(logging.NOTSET)} is called, it effectively removes this overriding level, so that logging output again depends on the effective levels of individual loggers.

logging.\texttt{addLevelName}(\texttt{lvl}, \texttt{levelName})

Associates level \texttt{lvl} with text \texttt{levelName} in an internal dictionary, which is used to map numeric levels to a textual representation, for example when a \texttt{Formatter} formats a message. This function can also be used to define your own levels. The only constraints
are that all levels used must be registered using this function, levels should be positive integers and they should increase in increasing order of severity.

**Note:** If you are thinking of defining your own levels, please see the section on Custom Levels.

logging. `getLevelName(lvl)`

Returns the textual representation of logging level `lvl`. If the level is one of the predefined levels **CRITICAL**, **ERROR**, **WARNING**, **INFO** or **DEBUG** then you get the corresponding string. If you have associated levels with names using `addLevelName()` then the name you have associated with `lvl` is returned. If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned. Otherwise, the string “Level %s” % lvl is returned.

**Note:** Integer levels should be used when e.g. setting levels on instances of **Logger** and handlers. This function is used to convert between an integer level and the level name displayed in the formatted log output by means of the `%s` format specifier (see LogRecord attributes).

logging. `makeLogRecord(attrdict)`

Creates and returns a new **LogRecord** instance whose attributes are defined by `attrdict`. This function is useful for taking a pickled **LogRecord** attribute dictionary, sent over a socket, and reconstituting it as a **LogRecord** instance at the receiving end.

logging. `basicConfig([**kwargs])`

Does basic configuration for the logging system by creating a **StreamHandler** with a default **Formatter** and adding it to the root
The functions `debug()`, `info()`, `warning()`, `error()` and `critical()` will call `basicConfig()` automatically if no handlers are defined for the root logger.

This function does nothing if the root logger already has handlers configured for it.

*Changed in version 2.4:* Formerly, `basicConfig()` did not take any keyword arguments.

**Note:** This function should be called from the main thread before other threads are started. In versions of Python prior to 2.7.1 and 3.2, if this function is called from multiple threads, it is possible (in rare circumstances) that a handler will be added to the root logger more than once, leading to unexpected results such as messages being duplicated in the log.

The following keyword arguments are supported.

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>filename</code></td>
<td>Specifies that a FileHandler be created, using the specified filename, rather than a StreamHandler.</td>
</tr>
<tr>
<td><code>filemode</code></td>
<td>Specifies the mode to open the file, if filename is specified (if filemode is unspecified, it defaults to ‘a’).</td>
</tr>
<tr>
<td><code>format</code></td>
<td>Use the specified format string for the handler.</td>
</tr>
<tr>
<td><code>datefmt</code></td>
<td>Use the specified date/time format.</td>
</tr>
<tr>
<td><code>level</code></td>
<td>Set the root logger level to the specified level.</td>
</tr>
<tr>
<td><code>stream</code></td>
<td>Use the specified stream to initialize the StreamHandler. Note that this argument is incompatible with ‘filename’ - if both are present, ‘stream’ is ignored.</td>
</tr>
</tbody>
</table>

`logging.shutdown()`
Informs the logging system to perform an orderly shutdown by flushing and closing all handlers. This should be called at application exit and no further use of the logging system should be made after this call.

logging.setLoggerClass(klass)
Tells the logging system to use the class klass when instantiating a logger. The class should define __init__() such that only a name argument is required, and the __init__() should call Logger.__init__(). This function is typically called before any loggers are instantiated by applications which need to use custom logger behavior.
The `captureWarnings()` function can be used to integrate logging with the `warnings` module.

```
logging.captureWarnings(capture)
```

This function is used to turn the capture of warnings by logging on and off.

If `capture` is `True`, warnings issued by the `warnings` module will be redirected to the logging system. Specifically, a warning will be formatted using `warnings.formatwarning()` and the resulting string logged to a logger named `'py.warnings'` with a severity of `WARNING`.

If `capture` is `False`, the redirection of warnings to the logging system will stop, and warnings will be redirected to their original destinations (i.e. those in effect before `captureWarnings(True)` was called).

See also:

**Module** `logging.config`
Configuration API for the logging module.

**Module** `logging.handlers`
Useful handlers included with the logging module.

**PEP 282 - A Logging System**
The proposal which described this feature for inclusion in the Python standard library.

**Original Python logging package**
This is the original source for the `logging` package. The version of the package available from this site is suitable for use with Python 1.5.2, 2.1.x and 2.2.x, which do not include the `logging` package in the standard library.
This section describes the API for configuring the logging module.

**Source code:** Lib/logging/config.py

---

**Important**

This page contains only reference information. For tutorials, please see:

- Basic Tutorial
- Advanced Tutorial
- Logging Cookbook
15.8.1. Configuration functions

The following functions configure the logging module. They are located in the `logging.config` module. Their use is optional — you can configure the logging module using these functions or by making calls to the main API (defined in `logging` itself) and defining handlers which are declared either in `logging` or `logging.handlers`.

`logging.config.dictConfig(config)`

Takes the logging configuration from a dictionary. The contents of this dictionary are described in Configuration dictionary schema below.

If an error is encountered during configuration, this function will raise a `ValueError`, `TypeError`, `AttributeError` or `ImportError` with a suitably descriptive message. The following is a (possibly incomplete) list of conditions which will raise an error:

- A `level` which is not a string or which is a string not corresponding to an actual logging level.
- A `propagate` value which is not a boolean.
- An id which does not have a corresponding destination.
- A non-existent handler id found during an incremental call.
- An invalid logger name.
- Inability to resolve to an internal or external object.

Parsing is performed by the `DictConfigurator` class, whose constructor is passed the dictionary used for configuration, and has a `configure()` method. The `logging.config` module has a callable attribute `dictConfigClass` which is initially set
to DictConfigurator. You can replace the value of dictConfigClass with a suitable implementation of your own.

dictConfig() calls dictConfigClass passing the specified dictionary, and then calls the configure() method on the returned object to put the configuration into effect:

```python
def dictConfig(config):
    dictConfigClass(config).configure()
```

For example, a subclass of DictConfigurator could call DictConfigurator.__init__() in its own __init__(), then set up custom prefixes which would be usable in the subsequent configure() call. dictConfigClass would be bound to this new subclass, and then dictConfig() could be called exactly as in the default, uncustomized state.

New in version 2.7.

logging.config.fileConfig(fname, defaults=None, disable_existing_loggers=True)

Reads the logging configuration from a configparser-format file named fname. The format of the file should be as described in Configuration file format. This function can be called several times from an application, allowing an end user to select from various pre-canned configurations (if the developer provides a mechanism to present the choices and load the chosen configuration).

Parameters:

- **defaults** – Defaults to be passed to the ConfigParser can be specified in this argument.
- **disable_existing_loggers** – If specified as False, loggers which exist when this call is made are left enabled. The default is True.
because this enables old behaviour in a backward-compatible way. This behaviour is to disable any existing loggers unless they or their ancestors are explicitly named in the logging configuration.

*Changed in version 2.6:* The `disable_existing_loggers` keyword argument was added. Previously, existing loggers were *always* disabled.

logging.config.**listen**(port=DEFAULT_LOGGING_CONFIG_PORT)

Starts up a socket server on the specified port, and listens for new configurations. If no port is specified, the module's default `DEFAULT_LOGGING_CONFIG_PORT` is used. Logging configurations will be sent as a file suitable for processing by `fileConfig()`. Returns a `Thread` instance on which you can call `start()` to start the server, and which you can `join()` when appropriate. To stop the server, call `stopListening()`.

To send a configuration to the socket, read in the configuration file and send it to the socket as a string of bytes preceded by a four-byte length string packed in binary using `struct.pack('>L', n)`.

**Note:** Because portions of the configuration are passed through `eval()`, use of this function may open its users to a security risk. While the function only binds to a socket on localhost, and so does not accept connections from remote machines, there are scenarios where untrusted code could be run under the account of the process which calls `listen()`. Specifically, if the process calling `listen()` runs on a multi-user machine where users cannot trust each other, then a malicious user could arrange to run essentially arbitrary code in a victim user’s process, simply by connecting to the victim’s `listen()`
socket and sending a configuration which runs whatever code the attacker wants to have executed in the victim's process. This is especially easy to do if the default port is used, but not hard even if a different port is used).

logging.config.stopListening()

Stops the listening server which was created with a call to listen(). This is typically called before calling join() on the return value from listen().
15.8.2. Configuration dictionary schema

Describing a logging configuration requires listing the various objects to create and the connections between them; for example, you may create a handler named ‘console’ and then say that the logger named ‘startup’ will send its messages to the ‘console’ handler. These objects aren’t limited to those provided by the logging module because you might write your own formatter or handler class. The parameters to these classes may also need to include external objects such as `sys.stderr`. The syntax for describing these objects and connections is defined in Object connections below.

15.8.2.1. Dictionary Schema Details

The dictionary passed to `dictConfig()` must contain the following keys:

- `version` - to be set to an integer value representing the schema version. The only valid value at present is 1, but having this key allows the schema to evolve while still preserving backwards compatibility.

All other keys are optional, but if present they will be interpreted as described below. In all cases below where a ‘configuring dict’ is mentioned, it will be checked for the special ‘()`’ key to see if a custom instantiation is required. If so, the mechanism described in User-defined objects below is used to create an instance; otherwise, the context is used to determine what to instantiate.

- `formatters` - the corresponding value will be a dict in which each key is a formatter id and each value is a dict describing how to configure the corresponding `Formatter` instance.
The configuring dict is searched for keys `format` and `datefmt` (with defaults of `None`) and these are used to construct a `Formatter` instance.

- `filters` - the corresponding value will be a dict in which each key is a filter id and each value is a dict describing how to configure the corresponding Filter instance.

The configuring dict is searched for the key `name` (defaulting to the empty string) and this is used to construct a `logging.Filter` instance.

- `handlers` - the corresponding value will be a dict in which each key is a handler id and each value is a dict describing how to configure the corresponding Handler instance.

The configuring dict is searched for the following keys:

- `class` (mandatory). This is the fully qualified name of the handler class.
- `level` (optional). The level of the handler.
- `formatter` (optional). The id of the formatter for this handler.
- `filters` (optional). A list of ids of the filters for this handler.

All other keys are passed through as keyword arguments to the handler’s constructor. For example, given the snippet:

```python
defining the content of a logging configuration file.

```
the handler with id **console** is instantiated as a `logging.StreamHandler`, using `sys.stdout` as the underlying stream. The handler with id **file** is instantiated as a `logging.handlers.RotatingFileHandler` with the keyword arguments `filename='logconfig.log', maxBytes=1024, backupCount=3`.

- **loggers** - the corresponding value will be a dict in which each key is a logger name and each value is a dict describing how to configure the corresponding Logger instance.

  The configuring dict is searched for the following keys:

  - `level` (optional). The level of the logger.
  - `propagate` (optional). The propagation setting of the logger.
  - `filters` (optional). A list of ids of the filters for this logger.
  - `handlers` (optional). A list of ids of the handlers for this logger.

  The specified loggers will be configured according to the level, propagation, filters and handlers specified.

- **root** - this will be the configuration for the root logger. Processing of the configuration will be as for any logger, except that the `propagate` setting will not be applicable.

- **incremental** - whether the configuration is to be interpreted as incremental to the existing configuration. This value defaults to `False`, which means that the specified configuration replaces the existing configuration with the same semantics as used by the existing `fileConfig()` API.
If the specified value is `True`, the configuration is processed as described in the section on Incremental Configuration.

- `disable_existing_loggers` - whether any existing loggers are to be disabled. This setting mirrors the parameter of the same name in `fileConfig()`. If absent, this parameter defaults to `True`. This value is ignored if `incremental` is `True`.

### 15.8.2.2. Incremental Configuration

It is difficult to provide complete flexibility for incremental configuration. For example, because objects such as filters and formatters are anonymous, once a configuration is set up, it is not possible to refer to such anonymous objects when augmenting a configuration.

Furthermore, there is not a compelling case for arbitrarily altering the object graph of loggers, handlers, filters, formatters at run-time, once a configuration is set up; the verbosity of loggers and handlers can be controlled just by setting levels (and, in the case of loggers, propagation flags). Changing the object graph arbitrarily in a safe way is problematic in a multi-threaded environment; while not impossible, the benefits are not worth the complexity it adds to the implementation.

Thus, when the `incremental` key of a configuration dict is present and is `True`, the system will completely ignore any `formatters` and `filters` entries, and process only the `level` settings in the `handlers` entries, and the `level` and `propagate` settings in the `loggers` and `root` entries.

Using a value in the configuration dict lets configurations to be sent over the wire as pickled dicts to a socket listener. Thus, the logging verbosity of a long-running application can be altered over time with
15.8.2.3. Object connections

The schema describes a set of logging objects - loggers, handlers, formatters, filters - which are connected to each other in an object graph. Thus, the schema needs to represent connections between the objects. For example, say that, once configured, a particular logger has attached to it a particular handler. For the purposes of this discussion, we can say that the logger represents the source, and the handler the destination, of a connection between the two. Of course in the configured objects this is represented by the logger holding a reference to the handler. In the configuration dict, this is done by giving each destination object an id which identifies it unambiguously, and then using the id in the source object's configuration to indicate that a connection exists between the source and the destination object with that id.

So, for example, consider the following YAML snippet:

```yaml
formatters:
  brief:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here

handlers:
  h1: #This is an id
    # configuration of handler with id 'h1' goes here
    formatter: brief
  h2: #This is another id
    # configuration of handler with id 'h2' goes here
    formatter: precise

loggers:
  foo.bar.baz:
    # other configuration for logger 'foo.bar.baz'
    handlers: [h1, h2]
```

(Note: YAML used here because it's a little more readable than the equivalent Python source form for the dictionary.)
The ids for loggers are the logger names which would be used programatically to obtain a reference to those loggers, e.g. foo.bar.baz. The ids for Formatters and Filters can be any string value (such as brief, precise above) and they are transient, in that they are only meaningful for processing the configuration dictionary and used to determine connections between objects, and are not persisted anywhere when the configuration call is complete.

The above snippet indicates that logger named foo.bar.baz should have two handlers attached to it, which are described by the handler ids h1 and h2. The formatter for h1 is that described by id brief, and the formatter for h2 is that described by id precise.

### 15.8.2.4. User-defined objects

The schema supports user-defined objects for handlers, filters and formatters. (Loggers do not need to have different types for different instances, so there is no support in this configuration schema for user-defined logger classes.)

Objects to be configured are described by dictionaries which detail their configuration. In some places, the logging system will be able to infer from the context how an object is to be instantiated, but when a user-defined object is to be instantiated, the system will not know how to do this. In order to provide complete flexibility for user-defined object instantiation, the user needs to provide a ‘factory’ - a callable which is called with a configuration dictionary and which returns the instantiated object. This is signalled by an absolute import path to the factory being made available under the special key '()' . Here’s a concrete example:

```python
formatters:
  brief:
    format: '%(message)s'
  default:
```

The above YAML snippet defines three formatters. The first, with id `brief`, is a standard `logging.Formatter` instance with the specified format string. The second, with id `default`, has a longer format and also defines the time format explicitly, and will result in a `logging.Formatter` initialized with those two format strings. Shown in Python source form, the `brief` and `default` formatters have configuration sub-dictionaries:

```yaml
{   'format' : '(%(message)s'
}
```

and:

```yaml
{   'format' : '(%asctime)s (%levelname)-8s (%name)-15s (%message)
    'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

respectively, and as these dictionaries do not contain the special key `'()'`, the instantiation is inferred from the context: as a result, standard `logging.Formatter` instances are created. The configuration sub-dictionary for the third formatter, with id `custom`, is:

```yaml
{   '()' : 'my.package.customFormatterFactory',
    'bar' : 'baz',
    'spam' : 99.9,
    'answer' : 42
}
```
and this contains the special key `'( )'`, which means that user-defined instantiation is wanted. In this case, the specified factory callable will be used. If it is an actual callable it will be used directly - otherwise, if you specify a string (as in the example) the actual callable will be located using normal import mechanisms. The callable will be called with the **remaining** items in the configuration sub-dictionary as keyword arguments. In the above example, the formatter with id `custom` will be assumed to be returned by the call:

```python
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

The key `'( )'` has been used as the special key because it is not a valid keyword parameter name, and so will not clash with the names of the keyword arguments used in the call. The `'( )'` also serves as a mnemonic that the corresponding value is a callable.

### 15.8.2.5. Access to external objects

There are times where a configuration needs to refer to objects external to the configuration, for example `sys.stderr`. If the configuration dict is constructed using Python code, this is straightforward, but a problem arises when the configuration is provided via a text file (e.g. JSON, YAML). In a text file, there is no standard way to distinguish `sys.stderr` from the literal string `sys.stderr`. To facilitate this distinction, the configuration system looks for certain special prefixes in string values and treat them specially. For example, if the literal string `'ext://sys.stderr'` is provided as a value in the configuration, then the `ext://` will be stripped off and the remainder of the value processed using normal import mechanisms.
The handling of such prefixes is done in a way analogous to protocol handling: there is a generic mechanism to look for prefixes which match the regular expression `^(?P<prefix>[a-z]+)://(?(P<suffix>.*))$` whereby, if the prefix is recognised, the suffix is processed in a prefix-dependent manner and the result of the processing replaces the string value. If the prefix is not recognised, then the string value will be left as-is.

### 15.8.2.6. Access to internal objects

As well as external objects, there is sometimes also a need to refer to objects in the configuration. This will be done implicitly by the configuration system for things that it knows about. For example, the string value `'DEBUG` for a level in a logger or handler will automatically be converted to the value `logging.DEBUG`, and the `handlers`, `filters` and `formatter` entries will take an object id and resolve to the appropriate destination object.

However, a more generic mechanism is needed for user-defined objects which are not known to the `logging` module. For example, consider `logging.handlers.MemoryHandler`, which takes a target argument which is another handler to delegate to. Since the system already knows about this class, then in the configuration, the given `target` just needs to be the object id of the relevant target handler, and the system will resolve to the handler from the id. If, however, a user defines a `my.package.MyHandler` which has an alternate handler, the configuration system would not know that the alternate referred to a handler. To cater for this, a generic resolution system allows the user to specify:

```yaml
handlers:
  file:
    # configuration of file handler goes here
```
custom:
    (): my.package.MyHandler
alternate: cfg://handlers.file

The literal string 'cfg://handlers.file' will be resolved in an analogous way to strings with the ext:// prefix, but looking in the configuration itself rather than the import namespace. The mechanism allows access by dot or by index, in a similar way to that provided by str.format. Thus, given the following snippet:

```python
def custom_handler_config():
    return ('cfg://handlers.file',)

my_handler = MyHandler()
alternate = cfg://handlers.file
```

in the configuration, the string 'cfg://handlers' would resolve to the dict with key handlers, the string 'cfg://handlers.email' would resolve to the dict with key email in the handlers dict, and so on. The string 'cfg://handlers.email.toaddrs[1]' would resolve to 'dev_team@domain.tld' and the string 'cfg://handlers.email.toaddrs[0]' would resolve to the value 'support_team@domain.tld'. The subject value could be accessed using either 'cfg://handlers.email.subject' or, equivalently, 'cfg://handlers.email[subject]'. The latter form only needs to be used if the key contains spaces or non-alphanumeric characters. If an index value consists only of decimal digits, access will be attempted using the corresponding integer value, falling back to the string value if needed.

Given a string 'cfg://handlers.myhandler.mykey.123', this will resolve to `config_dict['handlers']['myhandler']['mykey']['123']`. If the
string is specified as \texttt{cfg://handlers.myhandler.mykey[123]}, the system will attempt to retrieve the value from \texttt{config_dict['handlers']['myhandler']['mykey'][123]}, and fall back to \texttt{config_dict['handlers']['myhandler']['mykey']['123']} if that fails.

15.8.2.7. Import resolution and custom importers

Import resolution, by default, uses the built-in \texttt{__import__()} function to do its importing. You may want to replace this with your own importing mechanism: if so, you can replace the \texttt{importer} attribute of the \texttt{DictConfigurator} or its superclass, the \texttt{BaseConfigurator} class. However, you need to be careful because of the way functions are accessed from classes via descriptors. If you are using a Python callable to do your imports, and you want to define it at class level rather than instance level, you need to wrap it with \texttt{staticmethod()}. For example:

```python
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

You don't need to wrap with \texttt{staticmethod()} if you're setting the import callable on a configurator \textit{instance}. 
The configuration file format understood by `fileConfig()` is based on `configparser` functionality. The file must contain sections called `[loggers]`, `[handlers]` and `[formatters]` which identify by name the entities of each type which are defined in the file. For each such entity, there is a separate section which identifies how that entity is configured. Thus, for a logger named `log01` in the `[loggers]` section, the relevant configuration details are held in a section `[logger_log01]`. Similarly, a handler called `hand01` in the `[handlers]` section will have its configuration held in a section called `[handler_hand01]`, while a formatter called `form01` in the `[formatters]` section will have its configuration specified in a section called `[formatter_form01]`. The root logger configuration must be specified in a section called `[logger_root]`.

**Note:** The `fileConfig()` API is older than the `dictConfig()` API and does not provide functionality to cover certain aspects of logging. For example, you cannot configure `Filter` objects, which provide for filtering of messages beyond simple integer levels, using `fileConfig()`. If you need to have instances of `Filter` in your logging configuration, you will need to use `dictConfig()`. Note that future enhancements to configuration functionality will be added to `dictConfig()`, so it's worth considering transitioning to this newer API when it's convenient to do so.

Examples of these sections in the file are given below.

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
```
The root logger must specify a level and a list of handlers. An example of a root logger section is given below.

```
[logger_root]
level=NOTSET
handlers=hand01
```

The `level` entry can be one of `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL` or `NOTSET`. For the root logger only, `NOTSET` means that all messages will be logged. Level values are evaluated in the context of the `logging` package's namespace.

The `handlers` entry is a comma-separated list of handler names, which must appear in the `[handlers]` section. These names must appear in the `[handlers]` section and have corresponding sections in the configuration file.

For loggers other than the root logger, some additional information is required. This is illustrated by the following example.

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

The `level` and `handlers` entries are interpreted as for the root logger, except that if a non-root logger's level is specified as `NOTSET`, the system consults loggers higher up the hierarchy to determine the effective level of the logger. The `propagate` entry is set to 1 to indicate that messages must propagate to handlers higher up the
logger hierarchy from this logger, or 0 to indicate that messages are \textbf{not} propagated to handlers up the hierarchy. The \texttt{qualname} entry is the hierarchical channel name of the logger, that is to say the name used by the application to get the logger.

Sections which specify handler configuration are exemplified by the following.

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

The \texttt{class} entry indicates the handler's class (as determined by \texttt{eval()} in the \texttt{logging} package's namespace). The \texttt{level} is interpreted as for loggers, and \texttt{NOTSET} is taken to mean 'log everything'.

\textit{Changed in version 2.6}: Added support for resolving the handler's class as a dotted module and class name.

The \texttt{formatter} entry indicates the key name of the formatter for this handler. If blank, a default formatter (\texttt{logging._defaultFormatter}) is used. If a name is specified, it must appear in the \texttt{[formatters]} section and have a corresponding section in the configuration file.

The \texttt{args} entry, when \texttt{eval()}uated in the context of the \texttt{logging} package's namespace, is the list of arguments to the constructor for the handler class. Refer to the constructors for the relevant handlers, or to the examples below, to see how typical entries are constructed.

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')
```
[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogH

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
The \texttt{format} entry is the overall format string, and the \texttt{datefmt} entry is the \texttt{strftime}()-compatible date/time format string. If empty, the package substitutes ISO8601 format date/times, which is almost equivalent to specifying the date format string '\texttt{%Y-%m-%d %H:%M:%S}'. The ISO8601 format also specifies milliseconds, which are appended to the result of using the above format string, with a comma separator. An example time in ISO8601 format is 2003-01-23 00:29:50,411.

The \texttt{class} entry is optional. It indicates the name of the formatter's class (as a dotted module and class name.) This option is useful for instantiating a \texttt{Formatter} subclass. Subclasses of \texttt{Formatter} can present exception tracebacks in an expanded or condensed format.

\begin{quote}
\textbf{Note:} Due to the use of \texttt{eval()} as described above, there are potential security risks which result from using the \texttt{listen()} to send and receive configurations via sockets. The risks are limited to where multiple users with no mutual trust run code on the same machine; see the \texttt{listen()} documentation for more information.
\end{quote}

\textbf{See also:}

\begin{description}
\item[Module \texttt{logging}] API reference for the logging module.
\item[Module \texttt{logging.handlers}] Useful handlers included with the logging module.
\end{description}
The following useful handlers are provided in the package. Note that three of the handlers (`StreamHandler`, `FileHandler` and `NullHandler`) are actually defined in the `logging` module itself, but have been documented here along with the other handlers.
15.9.1. StreamHandler

The StreamHandler class, located in the core logging package, sends logging output to streams such as sys.stdout, sys.stderr or any file-like object (or, more precisely, any object which supports write() and flush() methods).

class logging.StreamHandler(stream=None)

Returns a new instance of the StreamHandler class. If stream is specified, the instance will use it for logging output; otherwise, sys.stderr will be used.

emit(record)

If a formatter is specified, it is used to format the record. The record is then written to the stream with a newline terminator. If exception information is present, it is formatted using traceback.print_exception() and appended to the stream.

flush()

Flushes the stream by calling its flush() method. Note that the close() method is inherited from Handler and so does no output, so an explicit flush() call may be needed at times.
The **FileHandler** class, located in the core **logging** package, sends logging output to a disk file. It inherits the output functionality from **StreamHandler**.

```python
class logging.FileHandler(filename, mode='a', encoding=None, delay=False)
```

Returns a new instance of the **FileHandler** class. The specified file is opened and used as the stream for logging. If `mode` is not specified, `'a'` is used. If `encoding` is not `None`, it is used to open the file with that encoding. If `delay` is true, then file opening is deferred until the first call to `emit()`. By default, the file grows indefinitely.

*Changed in version 2.6: delay was added.*

```python
close()
```

Closes the file.

```python
emit(record)
```

Outputs the record to the file.
15.9.3. NullHandler

*New in version 2.7.*

The **NullHandler** class, located in the core **logging** package, does not do any formatting or output. It is essentially a ‘no-op’ handler for use by library developers.

```python
class logging.NullHandler

Returns a new instance of the NullHandler class.
```

```python
emit(record)

This method does nothing.
```

```python
handle(record)

This method does nothing.
```

```python
createLock()

This method returns None for the lock, since there is no underlying I/O to which access needs to be serialized.
```

See [Configuring Logging for a Library](#) for more information on how to use **NullHandler**.
15.9.4. WatchedFileHandler

New in version 2.6.

The `WatchedFileHandler` class, located in the `logging.handlers` module, is a `FileHandler` which watches the file it is logging to. If the file changes, it is closed and reopened using the file name.

A file change can happen because of usage of programs such as `newsyslog` and `logrotate` which perform log file rotation. This handler, intended for use under Unix/Linux, watches the file to see if it has changed since the last emit. (A file is deemed to have changed if its device or inode have changed.) If the file has changed, the old file stream is closed, and the file opened to get a new stream.

This handler is not appropriate for use under Windows, because under Windows open log files cannot be moved or renamed - logging opens the files with exclusive locks - and so there is no need for such a handler. Furthermore, `ST_INO` is not supported under Windows; `stat()` always returns zero for this value.

class logging.handlers.WatchedFileHandler(filename[, mode[, encoding[, delay]]])

Returns a new instance of the `WatchedFileHandler` class. The specified file is opened and used as the stream for logging. If `mode` is not specified, 'a' is used. If `encoding` is not `None`, it is used to open the file with that encoding. If `delay` is true, then file opening is deferred until the first call to `emit()`. By default, the file grows indefinitely.

emit(record)

Outputs the record to the file, but first checks to see if the file has changed. If it has, the existing stream is flushed and
closed and the file opened again, before outputting the record to the file.
15.9.5. RotatingFileHandler

The RotatingFileHandler class, located in the logging.handlers module, supports rotation of disk log files.

```python
class logging.handlers.RotatingFileHandler(filename, mode='a', maxBytes=0, backupCount=0, encoding=None, delay=0)
```

Returns a new instance of the RotatingFileHandler class. The specified file is opened and used as the stream for logging. If mode is not specified, 'a' is used. If encoding is not None, it is used to open the file with that encoding. If delay is true, then file opening is deferred until the first call to `emit()`. By default, the file grows indefinitely.

You can use the maxBytes and backupCount values to allow the file to rollover at a predetermined size. When the size is about to be exceeded, the file is closed and a new file is silently opened for output. Rollover occurs whenever the current log file is nearly maxBytes in length; if either of maxBytes or backupCount is zero, rollover never occurs. If backupCount is non-zero, the system will save old log files by appending the extensions '.1', '.2' etc., to the filename. For example, with a backupCount of 5 and a base file name of `app.log`, you would get `app.log`, `app.log.1`, `app.log.2`, up to `app.log.5`. The file being written to is always `app.log`. When this file is filled, it is closed and renamed to `app.log.1`, and if files `app.log.1`, `app.log.2`, etc. exist, then they are renamed to `app.log.2`, `app.log.3` etc. respectively.

*Changed in version 2.6:* delay was added.

```python
def doRollover()
    Does a rollover, as described above.
```
emit(record)

Outputs the record to the file, catering for rollover as described previously.
15.9.6. TimedRotatingFileHandler

The TimedRotatingFileHandler class, located in the logging.handlers module, supports rotation of disk log files at certain timed intervals.

class logging.handlers.TimedRotatingFileHandler(filename, when='h', interval=1, backupCount=0, encoding=None, delay=False, utc=False)

Returns a new instance of the TimedRotatingFileHandler class. The specified file is opened and used as the stream for logging. On rotating it also sets the filename suffix. Rotating happens based on the product of when and interval.

You can use the when to specify the type of interval. The list of possible values is below. Note that they are not case sensitive.

<table>
<thead>
<tr>
<th>Value</th>
<th>Type of interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>'S'</td>
<td>Seconds</td>
</tr>
<tr>
<td>'M'</td>
<td>Minutes</td>
</tr>
<tr>
<td>'H'</td>
<td>Hours</td>
</tr>
<tr>
<td>'D'</td>
<td>Days</td>
</tr>
<tr>
<td>'W0'-'W6'</td>
<td>Weekday (0=Monday)</td>
</tr>
<tr>
<td>'midnight'</td>
<td>Roll over at midnight</td>
</tr>
</tbody>
</table>

When using weekday-based rotation, specify ‘W0’ for Monday, ‘W1’ for Tuesday, and so on up to ‘W6’ for Sunday. In this case, the value passed for interval isn’t used.

The system will save old log files by appending extensions to the filename. The extensions are date-and-time based, using the strftime format %Y-%m-%d_%H-%M-%S or a leading portion thereof, depending on the rollover interval.
When computing the next rollover time for the first time (when the handler is created), the last modification time of an existing log file, or else the current time, is used to compute when the next rotation will occur.

If the *utc* argument is true, times in UTC will be used; otherwise local time is used.

If *backupCount* is nonzero, at most *backupCount* files will be kept, and if more would be created when rollover occurs, the oldest one is deleted. The deletion logic uses the interval to determine which files to delete, so changing the interval may leave old files lying around.

If *delay* is true, then file opening is deferred until the first call to *emit()*.

*Changed in version 2.6: delay and utc were added.*

**doRollover()**

Does a rollover, as described above.

**emit(record)**

Outputs the record to the file, catering for rollover as described above.
15.9.7. SocketHandler

The `SocketHandler` class, located in the `logging.handlers` module, sends logging output to a network socket. The base class uses a TCP socket.

```python
class logging.handlers.SocketHandler(host, port)
    Returns a new instance of the `SocketHandler` class intended to communicate with a remote machine whose address is given by `host` and `port`.

close()
    Closes the socket.

emit()
    Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. If the connection was previously lost, re-establishes the connection. To unpickle the record at the receiving end into a `LogRecord`, use the `makeLogRecord()` function.

handleError()
    Handles an error which has occurred during `emit()`. The most likely cause is a lost connection. Closes the socket so that we can retry on the next event.

makeSocket()
    This is a factory method which allows subclasses to define the precise type of socket they want. The default implementation creates a TCP socket (`socket.SOCK_STREAM`).

makePickle(record)```
Pickles the record’s attribute dictionary in binary format with a length prefix, and returns it ready for transmission across the socket.

Note that pickles aren’t completely secure. If you are concerned about security, you may want to override this method to implement a more secure mechanism. For example, you can sign pickles using HMAC and then verify them on the receiving end, or alternatively you can disable unpickling of global objects on the receiving end.

**send**(*packet*)
Send a pickled string *packet* to the socket. This function allows for partial sends which can happen when the network is busy.

**createSocket**()
Tries to create a socket; on failure, uses an exponential back-off algorithm. On initial failure, the handler will drop the message it was trying to send. When subsequent messages are handled by the same instance, it will not try connecting until some time has passed. The default parameters are such that the initial delay is one second, and if after that delay the connection still can’t be made, the handler will double the delay each time up to a maximum of 30 seconds.

This behaviour is controlled by the following handler attributes:

- `retryStart` (initial delay, defaulting to 1.0 seconds).
- `retryFactor` (multiplier, defaulting to 2.0).
- `retryMax` (maximum delay, defaulting to 30.0 seconds).

This means that if the remote listener starts up after the handler has been used, you could lose messages (since the
handler won’t even attempt a connection until the delay has elapsed, but just silently drop messages during the delay period).
15.9.8. DatagramHandler

The **DatagramHandler** class, located in the `logging.handlers` module, inherits from **SocketHandler** to support sending logging messages over UDP sockets.

```python
class logging.handlers.DatagramHandler(host, port)
    Returns a new instance of the DatagramHandler class intended to communicate with a remote machine whose address is given by host and port.

    emit()
    Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. To unpickle the record at the receiving end into a LogRecord, use the makeLogRecord() function.

    makeSocket()
    The factory method of SocketHandler is here overridden to create a UDP socket (socket.SOCK_DGRAM).

    send(s)
    Send a pickled string to a socket.
```
The `SysLogHandler` class, located in the `logging.handlers` module, supports sending logging messages to a remote or local Unix syslog.

```python
class logging.handlers.SysLogHandler(address=('localhost', SYSLOG_UDP_PORT), facility=LOG_USER, socktype=socket.SOCK_DGRAM)
```

Returns a new instance of the `SysLogHandler` class intended to communicate with a remote Unix machine whose address is given by `address` in the form of a `(host, port)` tuple. If `address` is not specified, `('localhost', 514)` is used. The address is used to open a socket. An alternative to providing a `(host, port)` tuple is providing an address as a string, for example `'/dev/log'`. In this case, a Unix domain socket is used to send the message to the syslog. If `facility` is not specified, `LOG_USER` is used. The type of socket opened depends on the `socktype` argument, which defaults to `socket.SOCK_DGRAM` and thus opens a UDP socket. To open a TCP socket (for use with the newer syslog daemons such as rsyslog), specify a value of `socket.SOCK_STREAM`.

Note that if your server is not listening on UDP port 514, `SysLogHandler` may appear not to work. In that case, check what address you should be using for a domain socket - it's system dependent. For example, on Linux it's usually `'/dev/log'` but on OS/X it's `'/var/run/syslog'`. You'll need to check your platform and use the appropriate address (you may need to do this check at runtime if your application needs to run on several platforms). On Windows, you pretty much have to use the UDP option.

*Changed in version 2.7: socktype was added.*

`close()`
Closes the socket to the remote host.

**emit(record)**

The record is formatted, and then sent to the syslog server. If exception information is present, it is *not* sent to the server.

**encodePriority(facility, priority)**

Encodes the facility and priority into an integer. You can pass in strings or integers - if strings are passed, internal mapping dictionaries are used to convert them to integers.

The symbolic `LOG_` values are defined in `SysLogHandler` and mirror the values defined in the `sys/syslog.h` header file.

### Priorities

<table>
<thead>
<tr>
<th>Name (string)</th>
<th>Symbolic value</th>
</tr>
</thead>
<tbody>
<tr>
<td>alert</td>
<td>LOG_ALERT</td>
</tr>
<tr>
<td>crit or critical</td>
<td>LOG_CRIT</td>
</tr>
<tr>
<td>debug</td>
<td>LOG_DEBUG</td>
</tr>
<tr>
<td>emerg or panic</td>
<td>LOG_EMERG</td>
</tr>
<tr>
<td>err or error</td>
<td>LOG_ERR</td>
</tr>
<tr>
<td>info</td>
<td>LOG_INFO</td>
</tr>
<tr>
<td>notice</td>
<td>LOG_NOTICE</td>
</tr>
<tr>
<td>warn or warning</td>
<td>LOG_WARNING</td>
</tr>
</tbody>
</table>

### Facilities

<table>
<thead>
<tr>
<th>Name (string)</th>
<th>Symbolic value</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth</td>
<td>LOG_AUTH</td>
</tr>
<tr>
<td>authpriv</td>
<td>LOG_AUTHPRIV</td>
</tr>
<tr>
<td>cron</td>
<td>LOG_CRON</td>
</tr>
<tr>
<td>daemon</td>
<td>LOG_DAEMON</td>
</tr>
<tr>
<td>ftp</td>
<td>LOG_FTP</td>
</tr>
<tr>
<td>kern</td>
<td></td>
</tr>
<tr>
<td>Service</td>
<td>Priority</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------</td>
</tr>
<tr>
<td>lpr</td>
<td>LOG_LPR</td>
</tr>
<tr>
<td>mail</td>
<td>LOG_MAIL</td>
</tr>
<tr>
<td>news</td>
<td>LOG_NEWS</td>
</tr>
<tr>
<td>syslog</td>
<td>LOG_SYSLOG</td>
</tr>
<tr>
<td>user</td>
<td>LOG_USER</td>
</tr>
<tr>
<td>uucp</td>
<td>LOG_UUCP</td>
</tr>
<tr>
<td>local0</td>
<td>LOG_LOCAL0</td>
</tr>
<tr>
<td>local1</td>
<td>LOG_LOCAL1</td>
</tr>
<tr>
<td>local2</td>
<td>LOG_LOCAL2</td>
</tr>
<tr>
<td>local3</td>
<td>LOG_LOCAL3</td>
</tr>
<tr>
<td>local4</td>
<td>LOG_LOCAL4</td>
</tr>
<tr>
<td>local5</td>
<td>LOG_LOCAL5</td>
</tr>
<tr>
<td>local6</td>
<td>LOG_LOCAL6</td>
</tr>
<tr>
<td>local7</td>
<td>LOG_LOCAL7</td>
</tr>
</tbody>
</table>

**mapPriority(levelname)**

Maps a logging level name to a syslog priority name. You may need to override this if you are using custom levels, or if the default algorithm is not suitable for your needs. The default algorithm maps DEBUG, INFO, WARNING, ERROR and CRITICAL to the equivalent syslog names, and all other level names to 'warning'.
15.9.10. NTEventLogHandler

The NTEventLogHandler class, located in the logging.handlers module, supports sending logging messages to a local Windows NT, Windows 2000 or Windows XP event log. Before you can use it, you need Mark Hammond’s Win32 extensions for Python installed.

```python
class logging.handlers.NTEventLogHandler(appname,
dllname=None, logtype='Application')
```

Returns a new instance of the NTEventLogHandler class. The `appname` is used to define the application name as it appears in the event log. An appropriate registry entry is created using this name. The `dllname` should give the fully qualified pathname of a .dll or .exe which contains message definitions to hold in the log (if not specified, 'win32service.pyd' is used - this is installed with the Win32 extensions and contains some basic placeholder message definitions. Note that use of these placeholders will make your event logs big, as the entire message source is held in the log. If you want slimmer logs, you have to pass in the name of your own .dll or .exe which contains the message definitions you want to use in the event log). The `logtype` is one of 'Application', 'System' or 'Security', and defaults to 'Application'.

```python
close()
```

At this point, you can remove the application name from the registry as a source of event log entries. However, if you do this, you will not be able to see the events as you intended in the Event Log Viewer - it needs to be able to access the registry to get the .dll name. The current version does not do this.

```python
emit(record)
```
Determines the message ID, event category and event type, and then logs the message in the NT event log.

**getEventCategory**(record)

Returns the event category for the record. Override this if you want to specify your own categories. This version returns 0.

**getEventType**(record)

Returns the event type for the record. Override this if you want to specify your own types. This version does a mapping using the handler's typemap attribute, which is set up in `__init__()` to a dictionary which contains mappings for DEBUG, INFO, WARNING, ERROR and CRITICAL. If you are using your own levels, you will either need to override this method or place a suitable dictionary in the handler’s typemap attribute.

**getMessageID**(record)

Returns the message ID for the record. If you are using your own messages, you could do this by having the msg passed to the logger being an ID rather than a format string. Then, in here, you could use a dictionary lookup to get the message ID. This version returns 1, which is the base message ID in *win32service.pyd*. 
15.9.11. SMTPHandler

The `SMTPHandler` class, located in the `logging.handlers` module, supports sending logging messages to an email address via SMTP.

```python
class logging.handlers.SMTPHandler(mailhost, fromaddr, toaddr, subject, credentials=None, secure=None)
```

Returns a new instance of the `SMTPHandler` class. The instance is initialized with the from and to addresses and subject line of the email. The `toaddr` should be a list of strings. To specify a non-standard SMTP port, use the (host, port) tuple format for the `mailhost` argument. If you use a string, the standard SMTP port is used. If your SMTP server requires authentication, you can specify a (username, password) tuple for the `credentials` argument.

To specify the use of a secure protocol (TLS), pass in a tuple to the `secure` argument. This will only be used when authentication credentials are supplied. The tuple should be either an empty tuple, or a single-value tuple with the name of a keyfile, or a 2-value tuple with the names of the keyfile and certificate file. (This tuple is passed to the `smtplib.SMTP.starttls()` method.)

*Changed in version 2.6:* `credentials` was added.

*Changed in version 2.7:* `secure` was added.

```python
emit(record)
```

Formats the record and sends it to the specified addressee.

```python
getSubject(record)
```

If you want to specify a subject line which is record-dependent, override this method.
15.9.12. MemoryHandler

The MemoryHandler class, located in the logging.handlers module, supports buffering of logging records in memory, periodically flushing them to a target handler. Flushing occurs whenever the buffer is full, or when an event of a certain severity or greater is seen.

MemoryHandler is a subclass of the more general BufferingHandler, which is an abstract class. This buffers logging records in memory. Whenever each record is added to the buffer, a check is made by calling shouldFlush() to see if the buffer should be flushed. If it should, then flush() is expected to do the flushing.

```python
class logging.handlers.BufferingHandler(capacity):
    Initializes the handler with a buffer of the specified capacity.

def emit(record):
    Appends the record to the buffer. If shouldFlush() returns true, calls flush() to process the buffer.

def flush():
    You can override this to implement custom flushing behavior. This version just zaps the buffer to empty.

def shouldFlush(record):
    Returns true if the buffer is up to capacity. This method can be overridden to implement custom flushing strategies.
```

```python
class logging.handlers.MemoryHandler(capacity, flushLevel=ERROR, target=None):
    Returns a new instance of the MemoryHandler class. The instance is initialized with a buffer size of capacity. If flushLevel is not specified, ERROR is used. If no target is specified, the target will
need to be set using `setTarget()` before this handler does anything useful.

`close()`
Calls `flush()`, sets the target to `None` and clears the buffer.

`flush()`
For a `MemoryHandler`, flushing means just sending the buffered records to the target, if there is one. The buffer is also cleared when this happens. Override if you want different behavior.

`setTarget(target)`
Sets the target handler for this handler.

`shouldFlush(record)`
Checks for buffer full or a record at the `flushLevel` or higher.
15.9.13. HTTPHandler

The `HTTPHandler` class, located in the `logging.handlers` module, supports sending logging messages to a Web server, using either GET or POST semantics.

class logging.handlers.HTTPHandler(host, url, method='GET')

Returns a new instance of the `HTTPHandler` class. The `host` can be of the form `host:port`, should you need to use a specific port number.

`mapLogRecord(record)`

Provides a dictionary, based on `record`, which is to be URL-encoded and sent to the web server. The default implementation just returns `record.__dict__`. This method can be overridden if e.g. only a subset of `LogRecord` is to be sent to the web server, or if more specific customization of what's sent to the server is required.

`emit(record)`

Sends the record to the Web server as a URL-encoded dictionary. The `mapLogRecord()` method is used to convert the record to the dictionary to be sent.

**Note:** Since preparing a record for sending it to a Web server is not the same as a generic formatting operation, using `setFormatter()` to specify a `Formatter` for a `HTTPHandler` has no effect. Instead of calling `format()`, this handler calls `mapLogRecord()` and then `urllib.urlencode()` to encode the dictionary in a form suitable for sending to a Web server.

See also:
**Module** `logging`
API reference for the logging module.

**Module** `logging.config`
Configuration API for the logging module.
15.10. getpass — Portable password input

The `getpass` module provides two functions:

```python
getpass.getpass([prompt[, stream]])
```

Prompt the user for a password without echoing. The user is prompted using the string `prompt`, which defaults to `'Password:'`. On Unix, the prompt is written to the file-like object `stream`. `stream` defaults to the controlling terminal (`/dev/tty`) or if that is unavailable to `sys.stderr` (this argument is ignored on Windows).

If echo free input is unavailable `getpass()` falls back to printing a warning message to `stream` and reading from `sys.stdin` and issuing a `GetPassWarning`.

*Changed in version 2.5:* The `stream` parameter was added.

*Changed in version 2.6:* On Unix it defaults to using `/dev/tty` before falling back to `sys.stdin` and `sys.stderr`.

**Note:** If you call `getpass` from within IDLE, the input may be done in the terminal you launched IDLE from rather than the idle window itself.

An `UserWarning` subclass issued when password input may be echoed.

```python
getpass.getuser()
```

Return the “login name” of the user.
This function checks the environment variables `LOGNAME`, `USER`, `LNAME` and `USERNAME`, in order, and returns the value of the first one which is set to a non-empty string. If none are set, the login name from the password database is returned on systems which support the `pwd` module, otherwise, an exception is raised.
15.11. curses — Terminal handling for character-cell displays

*Changed in version 1.6:* Added support for the `ncurses` library and converted to a package.

The `curses` module provides an interface to the curses library, the de-facto standard for portable advanced terminal handling.

While curses is most widely used in the Unix environment, versions are available for DOS, OS/2, and possibly other systems as well. This extension module is designed to match the API of `ncurses`, an open-source curses library hosted on Linux and the BSD variants of Unix.

**Note:** Since version 5.4, the ncurses library decides how to interpret non-ASCII data using the `nl_langinfo` function. That means that you have to call `locale.setlocale()` in the application and encode Unicode strings using one of the system’s available encodings. This example uses the system’s default encoding:

```python
import locale
locale.setlocale(locale.LC_ALL, '')
code = locale.getpreferredencoding()
```

Then use `code` as the encoding for `str.encode()` calls.

**See also:**

*Module* `curses.ascii`

Utilities for working with ASCII characters, regardless of your locale settings.

*Module* `curses.panel`
A panel stack extension that adds depth to curses windows.

**Module** *curses.textpad*

Editable text widget for curses supporting *Emacs*-like bindings.

**Curses Programming with Python**

Tutorial material on using curses with Python, by Andrew Kuchling and Eric Raymond.

The Demo/curses/ directory in the Python source distribution contains some example programs using the curses bindings provided by this module.
15.11.1. Functions

The module `curses` defines the following exception:

```python
exception curses.error
```

Exception raised when a curses library function returns an error.

**Note:** Whenever `x` or `y` arguments to a function or a method are optional, they default to the current cursor location. Whenever `attr` is optional, it defaults to `A_NORMAL`.

The module `curses` defines the following functions:

```python
curses.baudrate()
```

Return the output speed of the terminal in bits per second. On software terminal emulators it will have a fixed high value. Included for historical reasons; in former times, it was used to write output loops for time delays and occasionally to change interfaces depending on the line speed.

```python
curses.beep()
```

Emit a short attention sound.

```python
curses.can_change_color()
```

Return `True` or `False`, depending on whether the programmer can change the colors displayed by the terminal.

```python
curses.cbreak()
```

Enter cbreak mode. In cbreak mode (sometimes called “rare” mode) normal tty line buffering is turned off and characters are available to be read one by one. However, unlike raw mode, special characters (interrupt, quit, suspend, and flow control) retain their effects on the tty driver and calling program. Calling
first raw() then cbreak() leaves the terminal in cbreak mode.

curses.color_content(color_number)
Return the intensity of the red, green, and blue (RGB) components in the color color_number, which must be between 0 and COLORS. A 3-tuple is returned, containing the R,G,B values for the given color, which will be between 0 (no component) and 1000 (maximum amount of component).

curses.color_pair(color_number)
Return the attribute value for displaying text in the specified color. This attribute value can be combined with A_STANDOUT, A_REVERSE, and the other A_* attributes. pair_number() is the counterpart to this function.

curses.curs_set(visibility)
Set the cursor state. visibility can be set to 0, 1, or 2, for invisible, normal, or very visible. If the terminal supports the visibility requested, the previous cursor state is returned; otherwise, an exception is raised. On many terminals, the “visible” mode is an underline cursor and the “very visible” mode is a block cursor.

curses.def_prog_mode()
Save the current terminal mode as the “program” mode, the mode when the running program is using curses. (Its counterpart is the “shell” mode, for when the program is not in curses.) Subsequent calls to reset_prog_mode() will restore this mode.

curses.def_shell_mode()
Save the current terminal mode as the “shell” mode, the mode when the running program is not using curses. (Its counterpart is the “program” mode, when the program is using curses capabilities.) Subsequent calls to reset_shell_mode() will restore this mode.
curses. **delay_output**(ms)

Insert an \textit{ms} millisecond pause in output.

curses. **doupdate()**

Update the physical screen. The curses library keeps two data structures, one representing the current physical screen contents and a virtual screen representing the desired next state. The **doupdate()** ground updates the physical screen to match the virtual screen.

The virtual screen may be updated by a **noutrefresh()** call after write operations such as **addstr()** have been performed on a window. The normal **refresh()** call is simply **noutrefresh()** followed by **doupdate()**; if you have to update multiple windows, you can speed performance and perhaps reduce screen flicker by issuing **noutrefresh()** calls on all windows, followed by a single **doupdate()**.

curses. **echo()**

Enter echo mode. In echo mode, each character input is echoed to the screen as it is entered.

curses. **endwin()**

De-initialize the library, and return terminal to normal status.

curses. **erasechar()**

Return the user's current erase character. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses. **filter()**

The **filter()** routine, if used, must be called before **initscr()** is called. The effect is that, during those calls, **LINES** is set to 1; the capabilities clear, cup, cud, cud1, cuu1, cuu, vpa are disabled;
and the home string is set to the value of cr. The effect is that the cursor is confined to the current line, and so are screen updates. This may be used for enabling character-at-a-time line editing without touching the rest of the screen.

**curses.** `flash()`

Flash the screen. That is, change it to reverse-video and then change it back in a short interval. Some people prefer such as 'visible bell' to the audible attention signal produced by `beep()`.

**curses.** `flushinp()`

Flush all input buffers. This throws away any typeahead that has been typed by the user and has not yet been processed by the program.

**curses.** `getmouse()`

After `getch()` returns `KEY_MOUSE` to signal a mouse event, this method should be call to retrieve the queued mouse event, represented as a 5-tuple `(id, x, y, z, bstate)`. `id` is an ID value used to distinguish multiple devices, and `x`, `y`, `z` are the event's coordinates. (`z` is currently unused.) `bstate` is an integer value whose bits will be set to indicate the type of event, and will be the bitwise OR of one or more of the following constants, where `n` is the button number from 1 to 4: `BUTTONn_PRESSED`, `BUTTONn_RELEASED`, `BUTTONn_CLICKED`, `BUTTONn_DOUBLE_CLICKED`, `BUTTONn_TRIPLE_CLICKED`, `BUTTON_SHIFT`, `BUTTON_CTRL`, `BUTTON_ALT`.

**curses.** `getsyx()`

Return the current coordinates of the virtual screen cursor in `y` and `x`. If `leaveok` is currently true, then `-1,-1` is returned.

**curses.** `getwin(file)`

Read window related data stored in the file by an earlier `putwin()` call. The routine then creates and initializes a new window using
that data, returning the new window object.

curses.**has_colors()**

Return **True** if the terminal can display colors; otherwise, return **False**.

curses.**has_ic()**

Return **True** if the terminal has insert- and delete-character capabilities. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

curses.**has_il()**

Return **True** if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

curses.**has_key(ch)**

Take a key value `ch`, and return **True** if the current terminal type recognizes a key with that value.

curses.**halfdelay(tenths)**

Used for half-delay mode, which is similar to cbreak mode in that characters typed by the user are immediately available to the program. However, after blocking for `tenths` tenths of seconds, an exception is raised if nothing has been typed. The value of `tenths` must be a number between 1 and 255. Use `nocbreak()` to leave half-delay mode.

curses.**init_color(color_number, r, g, b)**

Change the definition of a color, taking the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of `color_number` must be between 0 and `COLORS`. Each of `r, g, b`, must be a value
between 0 and 1000. When init_color() is used, all occurrences of that color on the screen immediately change to the new definition. This function is a no-op on most terminals; it is active only if can_change_color() returns 1.

**curses. init_pair(pair_number, fg, bg)**
Change the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of pair_number must be between 1 and COLOR_PAIRS - 1 (the 0 color pair is wired to white on black and cannot be changed). The value of fg and bg arguments must be between 0 and COLORS. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair are changed to the new definition.

**curses. initscr()**
Initialize the library. Return a WindowObject which represents the whole screen.

**Note:** If there is an error opening the terminal, the underlying curses library may cause the interpreter to exit.

**curses. is_term_resized(nlines, ncols)**
Return True if resize_term() would modify the window structure, False otherwise.

**curses. isendwin()**
Return True if endwin() has been called (that is, the curses library has been deinitialized).

**curses. keyname(k)**
Return the name of the key numbered k. The name of a key generating printable ASCII character is the key's character. The
name of a control-key combination is a two-character string consisting of a caret followed by the corresponding printable ASCII character. The name of an alt-key combination (128–255) is a string consisting of the prefix ‘M-‘ followed by the name of the corresponding ASCII character.

curses. `killchar()`

Return the user's current line kill character. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses. `longname()`

Return a string containing the terminfo long name field describing the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to `initscr()`.

curses. `meta(yes)`

If yes is 1, allow 8-bit characters to be input. If yes is 0, allow only 7-bit chars.

curses. `mouseinterval(interval)`

Set the maximum time in milliseconds that can elapse between press and release events in order for them to be recognized as a click, and return the previous interval value. The default value is 200 msec, or one fifth of a second.

curses. `mousemask(mousemask)`

Set the mouse events to be reported, and return a tuple (availmask, oldmask). `availmask` indicates which of the specified mouse events can be reported; on complete failure it returns 0. `oldmask` is the previous value of the given window's mouse event mask. If this function is never called, no mouse events are ever reported.
curses. **napms**(ms)

Sleep for *ms* milliseconds.

curses. **newpad**(nlines, ncols)

Create and return a pointer to a new pad data structure with the given number of lines and columns. A pad is returned as a window object.

A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (such as from scrolling or echoing of input) do not occur. The refresh() and noutrefresh() methods of a pad require 6 arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display. The arguments are pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol; the p arguments refer to the upper left corner of the pad region to be displayed and the s arguments define a clipping box on the screen within which the pad region is to be displayed.

curses. **newwin**(nlines, ncols)

curses. **newwin**(nlines, ncols, begin_y, begin_x)

Return a new window, whose left-upper corner is at (begin_y, begin_x), and whose height/width is nlines/ncols.

By default, the window will extend from the specified position to the lower right corner of the screen.

curses. **nl**()

Enter newline mode. This mode translates the return key into newline on input, and translates newline into return and line-feed on output. Newline mode is initially on.
curses. **nocbreak()**

Leave cbreak mode. Return to normal “cooked” mode with line buffering.

curses. **noecho()**

Leave echo mode. Echoing of input characters is turned off.

curses. **nonl()**

Leave newline mode. Disable translation of return into newline on input, and disable low-level translation of newline into newline/return on output (but this does not change the behavior of `addch('\n')`, which always does the equivalent of return and line feed on the virtual screen). With translation off, curses can sometimes speed up vertical motion a little; also, it will be able to detect the return key on input.

curses. **noqiflush()**

When the **noqiflush()** routine is used, normal flush of input and output queues associated with the INTR, QUIT and SUSP characters will not be done. You may want to call **noqiflush()** in a signal handler if you want output to continue as though the interrupt had not occurred, after the handler exits.

curses. **noraw()**

Leave raw mode. Return to normal “cooked” mode with line buffering.

curses. **pair_content(pair_number)**

Return a tuple (fg, bg) containing the colors for the requested color pair. The value of **pair_number** must be between 1 and **COLOR_PAIRS - 1**.

curses. **pair_number(attr)**

Return the number of the color-pair set by the attribute value **attr**.
color_pair() is the counterpart to this function.

curses.putp(string)
Equivalent to tputs(str, 1, putchar); emit the value of a specified terminfo capability for the current terminal. Note that the output of putp() always goes to standard output.

curses.qiflush([flag])
If flag is False, the effect is the same as calling noqiflush(). If flag is True, or no argument is provided, the queues will be flushed when these control characters are read.

curses.raw()
Enter raw mode. In raw mode, normal line buffering and processing of interrupt, quit, suspend, and flow control keys are turned off; characters are presented to curses input functions one by one.

curses.reset_prog_mode()
Restore the terminal to “program” mode, as previously saved by def_prog_mode().

curses.reset_shell_mode()
Restore the terminal to “shell” mode, as previously saved by def_shell_mode().

curses.resetty()
Restore the state of the terminal modes to what it was at the last call to savetty().

curses.resize_term(nlines, ncols)
Backend function used by resizeterm(), performing most of the work; when resizing the windows, resize_term() blank-fills the areas that are extended. The calling application should fill in
these areas with appropriate data. The `resize_term()` function attempts to resize all windows. However, due to the calling convention of pads, it is not possible to resize these without additional interaction with the application.

curses. **resizeterm**(nlines, ncols)
Resize the standard and current windows to the specified dimensions, and adjusts other bookkeeping data used by the curses library that record the window dimensions (in particular the SIGWINCH handler).

curses. **savetty**()
Save the current state of the terminal modes in a buffer, usable by `resetty()`.

curses. **setsyx**(y, x)
Set the virtual screen cursor to y, x. If y and x are both -1, then `leaveok` is set.

curses. **setupterm**([termstr, fd])
Initialize the terminal. `termstr` is a string giving the terminal name; if omitted, the value of the TERM environment variable will be used. `fd` is the file descriptor to which any initialization sequences will be sent; if not supplied, the file descriptor for `sys.stdout` will be used.

curses. **start_color**()
Must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after `initscr()`.

`start_color()` initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables in the `curses` module, `COLORS` and `COLOR_PAIRS`, containing the
maximum number of colors and color-pairs the terminal can support. It also restores the colors on the terminal to the values they had when the terminal was just turned on.

curses. termattrs()
Return a logical OR of all video attributes supported by the terminal. This information is useful when a curses program needs complete control over the appearance of the screen.

curses. termname()
Return the value of the environment variable TERM, truncated to 14 characters.

curses. tigetflag(capname)
Return the value of the Boolean capability corresponding to the terminfo capability name capname. The value -1 is returned if capname is not a Boolean capability, or 0 if it is canceled or absent from the terminal description.

curses. tigetnum(capname)
Return the value of the numeric capability corresponding to the terminfo capability name capname. The value -2 is returned if capname is not a numeric capability, or -1 if it is canceled or absent from the terminal description.

curses. tigetstr(capname)
Return the value of the string capability corresponding to the terminfo capability name capname. None is returned if capname is not a string capability, or is canceled or absent from the terminal description.

curses. tparm(str[,...])
Instantiate the string str with the supplied parameters, where str should be a parameterized string obtained from the terminfo
database. E.g. `tparm(tigetstr("cup"), 5, 3)` could result in `\033[6;4H`, the exact result depending on terminal type.

curses. **typeahead**(fd)
Specify that the file descriptor *fd* be used for typeahead checking. If *fd* is `-1`, then no typeahead checking is done.

The curses library does “line-breakout optimization” by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until refresh or doupdate is called again, allowing faster response to commands typed in advance. This function allows specifying a different file descriptor for typeahead checking.

curses. **unctrl**(ch)
Return a string which is a printable representation of the character *ch*. Control characters are displayed as a caret followed by the character, for example as `^C`. Printing characters are left as they are.

curses. **ungetch**(ch)
Push *ch* so the next `getch()` will return it.

**Note:** Only one *ch* can be pushed before `getch()` is called.

curses. **ungetmouse**(id, x, y, z, bstate)
Push a **KEY_MOUSE** event onto the input queue, associating the given state data with it.

curses. **use_env**(flag)
If used, this function should be called before `initscr()` or `newterm` are called. When *flag* is `False`, the values of lines and columns specified in the terminfo database will be used, even if environment variables `LINES` and `COLUMNS` (used by default) are
set, or if curses is running in a window (in which case default behavior would be to use the window size if **LINES** and **COLUMNS** are not set).

**curses.** **use_default_colors()**
Allow use of default values for colors on terminals supporting this feature. Use this to support transparency in your application. The default color is assigned to the color number -1. After calling this function, **init_pair(x, curses.COLOR_RED, -1)** initializes, for instance, color pair x to a red foreground color on the default background.

**curses.** **wrapper(func, ...)**
Initialize curses and call another callable object, **func**, which should be the rest of your curses-using application. If the application raises an exception, this function will restore the terminal to a sane state before re-raising the exception and generating a traceback. The callable object **func** is then passed the main window ‘stdscr’ as its first argument, followed by any other arguments passed to **wrapper()**. Before calling **func**, **wrapper()** turns on cbreak mode, turns off echo, enables the terminal keypad, and initializes colors if the terminal has color support. On exit (whether normally or by exception) it restores cooked mode, turns on echo, and disables the terminal keypad.
15.11.2. Window Objects

Window objects, as returned by `initscr()` and `newwin()` above, have the following methods:

```python
window. addch(ch[, attr])
window. addch(y, x, ch[, attr])
```

**Note:** A character means a C character (an ASCII code), rather than a Python character (a string of length 1). (This note is true whenever the documentation mentions a character.) The built-in `ord()` is handy for conveying strings to codes.

Paint character `ch` at `(y, x)` with attributes `attr`, overwriting any character previously painter at that location. By default, the character position and attributes are the current settings for the window object.

```python
window. addnstr(str, n[, attr])
window. addnstr(y, x, str, n[, attr])
```

Paint at most `n` characters of the string `str` at `(y, x)` with attributes `attr`, overwriting anything previously on the display.

```python
window. addstr(str[, attr])
window. addstr(y, x, str[, attr])
```

Paint the string `str` at `(y, x)` with attributes `attr`, overwriting anything previously on the display.

```python
window. attroff(attr)
```

Remove attribute `attr` from the “background” set applied to all writes to the current window.
window. `attron(attr)`
Add attribute `attr` from the “background” set applied to all writes to the current window.

window. `attrset(attr)`
Set the “background” set of attributes to `attr`. This set is initially 0 (no attributes).

window. `bkgd(ch[, attr])`
Set the background property of the window to the character `ch`, with attributes `attr`. The change is then applied to every character position in that window:

- The attribute of every character in the window is changed to the new background attribute.
- Wherever the former background character appears, it is changed to the new background character.

window. `bkgdset(ch[, attr])`
Set the window’s background. A window’s background consists of a character and any combination of attributes. The attribute part of the background is combined (OR’ed) with all non-blank characters that are written into the window. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations.

window. `border([ls[, rs[, ts[, bs[, tl[, tr[, bl[, br]]]]]])]
Draw a border around the edges of the window. Each parameter specifies the character to use for a specific part of the border; see the table below for more details. The characters can be specified as integers or as one-character strings.
**Note:** A \( \theta \) value for any parameter will cause the default character to be used for that parameter. Keyword parameters can *not* be used. The defaults are listed in this table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ls )</td>
<td>Left side</td>
<td>ACS_VLINE</td>
</tr>
<tr>
<td>( rs )</td>
<td>Right side</td>
<td>ACS_VLINE</td>
</tr>
<tr>
<td>( ts )</td>
<td>Top</td>
<td>ACS_HLINE</td>
</tr>
<tr>
<td>( bs )</td>
<td>Bottom</td>
<td>ACS_HLINE</td>
</tr>
<tr>
<td>( tl )</td>
<td>Upper-left corner</td>
<td>ACS_ULCORNER</td>
</tr>
<tr>
<td>( tr )</td>
<td>Upper-right corner</td>
<td>ACS_URCORNER</td>
</tr>
<tr>
<td>( bl )</td>
<td>Bottom-left corner</td>
<td>ACS_LLCORNER</td>
</tr>
<tr>
<td>( br )</td>
<td>Bottom-right corner</td>
<td>ACS_LRCORNER</td>
</tr>
</tbody>
</table>

\texttt{window.box([vertch, horch])}

Similar to \texttt{border()}, but both \( ls \) and \( rs \) are \textit{vertch} and both \( ts \) and \( bs \) are \textit{horch}. The default corner characters are always used by this function.

\texttt{window.chgat(attr)}
\texttt{window.chgat(num, attr)}
\texttt{window.chgat(y, x, attr)}
\texttt{window.chgat(y, x, num, attr)}

Set the attributes of \textit{num} characters at the current cursor position, or at position \((y, x)\) if supplied. If no value of \textit{num} is given or \textit{num} = -1, the attribute will be set on all the characters to the end of the line. This function does not move the cursor. The changed line will be touched using the \texttt{touchline()} method so that the contents will be redisplayed by the next window refresh.

\texttt{window.clear()}

Like \texttt{erase()}, but also cause the whole window to be repainted upon next call to \texttt{refresh()}. 
window. `clearok(yes)`
If yes is 1, the next call to `refresh()` will clear the window completely.

window. `clrtobot()`
Erase from cursor to the end of the window: all lines below the cursor are deleted, and then the equivalent of `clrtoeol()` is performed.

window. `clrtoeol()`
Erase from cursor to the end of the line.

window. `cursyncup()`
Update the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

window. `delch([y, x])`
Delete any character at (y, x).

window. `deleteln()`
Delete the line under the cursor. All following lines are moved up by one line.

window. `derwin(begin_y, begin_x)`
window. `derwin(nlines, ncols, begin_y, begin_x)`
An abbreviation for “derive window”, `derwin()` is the same as calling `subwin()`, except that `begin_y` and `begin_x` are relative to the origin of the window, rather than relative to the entire screen. Return a window object for the derived window.

window. `echochar(ch[, attr])`
Add character `ch` with attribute `attr`, and immediately call `refresh()` on the window.
window. `enclose(y, x)`
Test whether the given pair of screen-relative character-cell coordinates are enclosed by the given window, returning True or False. It is useful for determining what subset of the screen windows enclose the location of a mouse event.

window. `erase()`
Clear the window.

window. `getbegyx()`
Return a tuple $(y, x)$ of co-ordinates of upper-left corner.

window. `getbkgd()`
Return the given window’s current background character/attribute pair.

window. `getch([y, x])`
Get a character. Note that the integer returned does not have to be in ASCII range: function keys, keypad keys and so on return numbers higher than 256. In no-delay mode, -1 is returned if there is no input, else `getch()` waits until a key is pressed.

window. `getkey([y, x])`
Get a character, returning a string instead of an integer, as `getch()` does. Function keys, keypad keys and so on return a multibyte string containing the key name. In no-delay mode, an exception is raised if there is no input.

window. `getmaxyx()`
Return a tuple $(y, x)$ of the height and width of the window.

window. `getparyx()`
Return the beginning coordinates of this window relative to its parent window into two integer variables y and x. Return -1, -1 if
this window has no parent.

`window.getstr([y, x])`

Read a string from the user, with primitive line editing capacity.

`window.getyx()`

Return a tuple \((y, x)\) of current cursor position relative to the window's upper-left corner.

`window.hline(ch, n)`

`window.hline(y, x, ch, n)`

Display a horizontal line starting at \((y, x)\) with length \(n\) consisting of the character \(ch\).

`window.idcok(flag)`

If \(flag\) is `False`, curses no longer considers using the hardware insert/delete character feature of the terminal; if \(flag\) is `True`, use of character insertion and deletion is enabled. When curses is first initialized, use of character insert/delete is enabled by default.

`window.idlok(yes)`

If called with `yes` equal to 1, curses will try and use hardware line editing facilities. Otherwise, line insertion/deletion are disabled.

`window.immedok(flag)`

If \(flag\) is `True`, any change in the window image automatically causes the window to be refreshed; you no longer have to call `refresh()` yourself. However, it may degrade performance considerably, due to repeated calls to `wrefresh`. This option is disabled by default.

`window.inch([y, x])`

Return the character at the given position in the window. The
bottom 8 bits are the character proper, and upper bits are the attributes.

`window.insch(ch[, attr])`
`window.insch(y, x, ch[, attr])`

Paint character `ch` at `(y, x)` with attributes `attr`, moving the line from position `x` right by one character.

`window.insdelln(nlines)`

Insert `nlines` lines into the specified window above the current line. The `nlines` bottom lines are lost. For negative `nlines`, delete `nlines` lines starting with the one under the cursor, and move the remaining lines up. The bottom `nlines` lines are cleared. The current cursor position remains the same.

`window.insert1n()`

Insert a blank line under the cursor. All following lines are moved down by one line.

`window.insnstr(str, n[, attr])`
`window.insnstr(y, x, str, n[, attr])`

Insert a character string (as many characters as will fit on the line) before the character under the cursor, up to `n` characters. If `n` is zero or negative, the entire string is inserted. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to `y, x`, if specified).

`window.insstr(str[, attr])`
`window.insstr(y, x, str[, attr])`

Insert a character string (as many characters as will fit on the line) before the character under the cursor. All characters to the right of the cursor are shifted right, with the rightmost characters
on the line being lost. The cursor position does not change (after moving to \textit{y, x}, if specified).

\texttt{window.instr([n])}

\texttt{window.instr(y, x[, n])}

Return a string of characters, extracted from the window starting at the current cursor position, or at \textit{y, x} if specified. Attributes are stripped from the characters. If \textit{n} is specified, \texttt{instr()} returns a string at most \textit{n} characters long (exclusive of the trailing NUL).

\texttt{window.is_linetouched(line)}

Return \texttt{True} if the specified line was modified since the last call to \texttt{refresh();} otherwise return \texttt{False}. Raise a \texttt{curses.error} exception if \textit{line} is not valid for the given window.

\texttt{window.is_wintouched()}

Return \texttt{True} if the specified window was modified since the last call to \texttt{refresh();} otherwise return \texttt{False}.

\texttt{window.keypad(yes)}

If \textit{yes} is 1, escape sequences generated by some keys (keypad, function keys) will be interpreted by \texttt{curses}. If \textit{yes} is 0, escape sequences will be left as is in the input stream.

\texttt{window.leaveok(yes)}

If \textit{yes} is 1, cursor is left where it is on update, instead of being at “cursor position.” This reduces cursor movement where possible. If possible the cursor will be made invisible.

If \textit{yes} is 0, cursor will always be at “cursor position” after an update.

\texttt{window.move(new_y, new_x)}

Move cursor to \texttt{(new_y, new_x)}.
window. **mvderwin**(\(y, x\))
Move the window inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

window. **mvwin**(\(new\_y, new\_x\))
Move the window so its upper-left corner is at \((new\_y, new\_x)\).

window. **nodelay**(yes)
If yes is 1, `getch()` will be non-blocking.

window. **notimeout**(yes)
If yes is 1, escape sequences will not be timed out.

If yes is 0, after a few milliseconds, an escape sequence will not be interpreted, and will be left in the input stream as is.

window. **noutrefresh**( )
Mark for refresh but wait. This function updates the data structure representing the desired state of the window, but does not force an update of the physical screen. To accomplish that, call `doupdate()`.

window. **overlay**(destwin[, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol])
Overlay the window on top of destwin. The windows need not be the same size, only the overlapping region is copied. This copy is non-destructive, which means that the current background character does not overwrite the old contents of destwin.

To get fine-grained control over the copied region, the second form of `overlay()` can be used. `sminrow` and `smincol` are the upper-left coordinates of the source window, and the other
variables mark a rectangle in the destination window.

```plaintext
window. overwrite(destwin[, sminrow, smincol, dminrow, dmincol,
dmaxrow, dmaxcol])
```

Overwrite the window on top of `destwin`. The windows need not be the same size, in which case only the overlapping region is copied. This copy is destructive, which means that the current background character overwrites the old contents of `destwin`.

To get fine-grained control over the copied region, the second form of `overwrite()` can be used. `sminrow` and `smincol` are the upper-left coordinates of the source window, the other variables mark a rectangle in the destination window.

```plaintext
window. putwin(file)
```

Write all data associated with the window into the provided file object. This information can be later retrieved using the `getwin()` function.

```plaintext
window. redrawln(beg, num)
```

Indicate that the `num` screen lines, starting at line `beg`, are corrupted and should be completely redrawn on the next `refresh()` call.

```plaintext
window. redrawwin()
```

Touch the entire window, causing it to be completely redrawn on the next `refresh()` call.

```plaintext
window. refresh([pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol])
```

Update the display immediately (sync actual screen with previous drawing/deleting methods).

The 6 optional arguments can only be specified when the window
is a pad created with `newpad()` . The additional parameters are needed to indicate what part of the pad and screen are involved. `pminrow` and `pmincol` specify the upper left-hand corner of the rectangle to be displayed in the pad. `sminrow`, `smincol`, `smaxrow`, and `smaxcol` specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of `pminrow`, `pmincol`, `sminrow`, or `smincol` are treated as if they were zero.

```
window. resize(nlines, ncols)
```

Reallocate storage for a curses window to adjust its dimensions to the specified values. If either dimension is larger than the current values, the window’s data is filled with blanks that have the current background rendition (as set by `bkgdset()`) merged into them.

```
window. scroll([lines=1])
```

Scroll the screen or scrolling region upward by `lines` lines.

```
window. scrollok(flag)
```

Control what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If `flag` is false, the cursor is left on the bottom line. If `flag` is true, the window is scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call `idlok()`.

```
window. setscrreg(top, bottom)
```

Set the scrolling region from line `top` to line `bottom`. All scrolling actions will take place in this region.
`window. standend()`

Turn off the standout attribute. On some terminals this has the side effect of turning off all attributes.

`window. standout()`

Turn on attribute `A_STANDOUT`.

`window. subpad(begin_y, begin_x)`

`window. subpad(nlines, ncols, begin_y, begin_x)`

Return a sub-window, whose upper-left corner is at `(begin_y, begin_x)`, and whose width/height is `ncols/nlines`.

`window. subwin(begin_y, begin_x)`

`window. subwin(nlines, ncols, begin_y, begin_x)`

Return a sub-window, whose upper-left corner is at `(begin_y, begin_x)`, and whose width/height is `ncols/nlines`.

By default, the sub-window will extend from the specified position to the lower right corner of the window.

`window. syncdown()`

Touch each location in the window that has been touched in any of its ancestor windows. This routine is called by `refresh()`, so it should almost never be necessary to call it manually.

`window. syncok(flag)`

If called with `flag` set to `True`, then `syncup()` is called automatically whenever there is a change in the window.

`window. syncup()`

Touch all locations in ancestors of the window that have been changed in the window.

`window. timeout(delay)`

Set blocking or non-blocking read behavior for the window. If
delay is negative, blocking read is used (which will wait
indefinitely for input). If delay is zero, then non-blocking read is
used, and -1 will be returned by getch() if no input is waiting. If
delay is positive, then getch() will block for delay milliseconds,
and return -1 if there is still no input at the end of that time.

window. touchline(start, count[, changed])
Pretend count lines have been changed, starting with line start. If
changed is supplied, it specifies whether the affected lines are
marked as having been changed (changed=1) or unchanged
(changed=0).

window. touchwin()
Pretend the whole window has been changed, for purposes of
drawing optimizations.

window. untouchwin()
Mark all lines in the window as unchanged since the last call to
refresh().

window. vline(ch, n)
window. vline(y, x, ch, n)
Display a vertical line starting at (y, x) with length n consisting
of the character ch.
15.11.3. Constants

The `curses` module defines the following data members:

`curses.ERR`
Some curses routines that return an integer, such as `getch()`, return `ERR` upon failure.

`curses.OK`
Some curses routines that return an integer, such as `napms()`, return `OK` upon success.

`curses.version`
A string representing the current version of the module. Also available as `__version__`.

Several constants are available to specify character cell attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_ALTCHARSET</td>
<td>Alternate character set mode.</td>
</tr>
<tr>
<td>A_BLINK</td>
<td>Blink mode.</td>
</tr>
<tr>
<td>A_BOLD</td>
<td>Bold mode.</td>
</tr>
<tr>
<td>A_DIM</td>
<td>Dim mode.</td>
</tr>
<tr>
<td>A_NORMAL</td>
<td>Normal attribute.</td>
</tr>
<tr>
<td>A_REVERSE</td>
<td>Reverse background and foreground colors.</td>
</tr>
<tr>
<td>A_STANDOUT</td>
<td>Standout mode.</td>
</tr>
<tr>
<td>A_UNDERLINE</td>
<td>Underline mode.</td>
</tr>
</tbody>
</table>

Keys are referred to by integer constants with names starting with `KEY_`. The exact keycaps available are system dependent.

<table>
<thead>
<tr>
<th>Key constant</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_MIN</td>
<td>Minimum key value</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>KEY_BREAK</td>
<td>Break key (unreliable)</td>
</tr>
<tr>
<td>KEY_DOWN</td>
<td>Down-arrow</td>
</tr>
<tr>
<td>KEY_UP</td>
<td>Up-arrow</td>
</tr>
<tr>
<td>KEY_LEFT</td>
<td>Left-arrow</td>
</tr>
<tr>
<td>KEY_RIGHT</td>
<td>Right-arrow</td>
</tr>
<tr>
<td>KEY_HOME</td>
<td>Home key (upward+left arrow)</td>
</tr>
<tr>
<td>KEY_BACKSPACE</td>
<td>Backspace (unreliable)</td>
</tr>
<tr>
<td>KEY_F0</td>
<td>Function keys. Up to 64 function keys are supported.</td>
</tr>
<tr>
<td>KEY_Fn</td>
<td>Value of function key $n$</td>
</tr>
<tr>
<td>KEY_DL</td>
<td>Delete line</td>
</tr>
<tr>
<td>KEY_IL</td>
<td>Insert line</td>
</tr>
<tr>
<td>KEY_DC</td>
<td>Delete character</td>
</tr>
<tr>
<td>KEY_IC</td>
<td>Insert char or enter insert mode</td>
</tr>
<tr>
<td>KEY_EIC</td>
<td>Exit insert char mode</td>
</tr>
<tr>
<td>KEY_CLEAR</td>
<td>Clear insert char mode</td>
</tr>
<tr>
<td>KEY_EOS</td>
<td>Clear to end of screen</td>
</tr>
<tr>
<td>KEY_EOL</td>
<td>Clear to end of line</td>
</tr>
<tr>
<td>KEY_SF</td>
<td>Scroll 1 line forward</td>
</tr>
<tr>
<td>KEY_SR</td>
<td>Scroll 1 line backward (reverse)</td>
</tr>
<tr>
<td>KEY_NPAGE</td>
<td>Next page</td>
</tr>
<tr>
<td>KEY_PPAGE</td>
<td>Previous page</td>
</tr>
<tr>
<td>KEY_STAB</td>
<td>Set tab</td>
</tr>
<tr>
<td>KEY_CTAB</td>
<td>Clear tab</td>
</tr>
<tr>
<td>KEY_CATAB</td>
<td>Clear all tabs</td>
</tr>
<tr>
<td>KEY_ENTER</td>
<td>Enter or send (unreliable)</td>
</tr>
<tr>
<td>KEY_SRESET</td>
<td>Soft (partial) reset (unreliable)</td>
</tr>
<tr>
<td>KEY_RESET</td>
<td>Reset or hard reset (unreliable)</td>
</tr>
<tr>
<td>KEY_PRINT</td>
<td>Print</td>
</tr>
<tr>
<td>KEY_LL</td>
<td>Home down or bottom (lower left)</td>
</tr>
<tr>
<td>KEY_A1</td>
<td>Upper left of keypad</td>
</tr>
<tr>
<td>KEY_A3</td>
<td>Upper right of keypad</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>KEY_B2</td>
<td>Center of keypad</td>
</tr>
<tr>
<td>KEY_C1</td>
<td>Lower left of keypad</td>
</tr>
<tr>
<td>KEY_C3</td>
<td>Lower right of keypad</td>
</tr>
<tr>
<td>KEY_BTAB</td>
<td>Back tab</td>
</tr>
<tr>
<td>KEY_BEG</td>
<td>Beg (beginning)</td>
</tr>
<tr>
<td>KEY_CANCEL</td>
<td>Cancel</td>
</tr>
<tr>
<td>KEY_CLOSE</td>
<td>Close</td>
</tr>
<tr>
<td>KEY_COMMAND</td>
<td>Cmd (command)</td>
</tr>
<tr>
<td>KEY_COPY</td>
<td>Copy</td>
</tr>
<tr>
<td>KEY_CREATE</td>
<td>Create</td>
</tr>
<tr>
<td>KEY_END</td>
<td>End</td>
</tr>
<tr>
<td>KEY_EXIT</td>
<td>Exit</td>
</tr>
<tr>
<td>KEY_FIND</td>
<td>Find</td>
</tr>
<tr>
<td>KEY_HELP</td>
<td>Help</td>
</tr>
<tr>
<td>KEY_MARK</td>
<td>Mark</td>
</tr>
<tr>
<td>KEY_MESSAGE</td>
<td>Message</td>
</tr>
<tr>
<td>KEY_MOVE</td>
<td>Move</td>
</tr>
<tr>
<td>KEY_NEXT</td>
<td>Next</td>
</tr>
<tr>
<td>KEY_OPEN</td>
<td>Open</td>
</tr>
<tr>
<td>KEY_OPTIONS</td>
<td>Options</td>
</tr>
<tr>
<td>KEY_PREVIOUS</td>
<td>Prev (previous)</td>
</tr>
<tr>
<td>KEY_REDO</td>
<td>Redo</td>
</tr>
<tr>
<td>KEY_REFERENCE</td>
<td>Ref (reference)</td>
</tr>
<tr>
<td>KEY_REFRESH</td>
<td>Refresh</td>
</tr>
<tr>
<td>KEY_REPLACE</td>
<td>Replace</td>
</tr>
<tr>
<td>KEY_RESTART</td>
<td>Restart</td>
</tr>
<tr>
<td>KEY_RESUME</td>
<td>Resume</td>
</tr>
<tr>
<td>KEY_SAVE</td>
<td>Save</td>
</tr>
<tr>
<td>KEY_SBEG</td>
<td>Shifted Beg (beginning)</td>
</tr>
<tr>
<td>KEY_SCANCEL</td>
<td>Shifted Cancel</td>
</tr>
<tr>
<td>KEY_SCOMMAND</td>
<td>Shifted Command</td>
</tr>
<tr>
<td>Key Code</td>
<td>Action Description</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>KEY_SCOPY</td>
<td>Shifted Copy</td>
</tr>
<tr>
<td>KEY_SCREATE</td>
<td>Shifted Create</td>
</tr>
<tr>
<td>KEY_SDC</td>
<td>Shifted Delete char</td>
</tr>
<tr>
<td>KEY(SDL)</td>
<td>Shifted Delete line</td>
</tr>
<tr>
<td>KEY_SELECT</td>
<td>Select</td>
</tr>
<tr>
<td>KEY_SEND</td>
<td>Shifted End</td>
</tr>
<tr>
<td>KEY_SEOL</td>
<td>Shifted Clear line</td>
</tr>
<tr>
<td>KEY_SEXIT</td>
<td>Shifted Dxit</td>
</tr>
<tr>
<td>KEY_SFIN</td>
<td>Shifted Find</td>
</tr>
<tr>
<td>KEY_SHELP</td>
<td>Shifted Help</td>
</tr>
<tr>
<td>KEY_SHOME</td>
<td>Shifted Home</td>
</tr>
<tr>
<td>KEY_SIC</td>
<td>Shifted Input</td>
</tr>
<tr>
<td>KEY_SLEFT</td>
<td>Shifted Left arrow</td>
</tr>
<tr>
<td>KEY_SMES</td>
<td>Shifted Message</td>
</tr>
<tr>
<td>KEY_SMOVE</td>
<td>Shifted Move</td>
</tr>
<tr>
<td>KEY_SNEXT</td>
<td>Shifted Next</td>
</tr>
<tr>
<td>KEY&amp;SOP</td>
<td>Shifted Options</td>
</tr>
<tr>
<td>KEY_SPREV</td>
<td>Shifted Prev</td>
</tr>
<tr>
<td>KEY_SPRINT</td>
<td>Shifted Print</td>
</tr>
<tr>
<td>KEY_SREDO</td>
<td>Shifted Redo</td>
</tr>
<tr>
<td>KEY_SREPL</td>
<td>Shifted Replace</td>
</tr>
<tr>
<td>KEY_SRI</td>
<td>Shifted Right arrow</td>
</tr>
<tr>
<td>KEY_SRUS</td>
<td>Shifted Resume</td>
</tr>
<tr>
<td>KEY_SSAVE</td>
<td>Shifted Save</td>
</tr>
<tr>
<td>KEY_SSUSP</td>
<td>Shifted Suspend</td>
</tr>
<tr>
<td>KEY_SUND</td>
<td>Shifted Undo</td>
</tr>
<tr>
<td>KEY_SUSP</td>
<td>Suspend</td>
</tr>
<tr>
<td>KEY_UND</td>
<td>Undo</td>
</tr>
<tr>
<td>KEY_MOUSE</td>
<td>Mouse event has occurred</td>
</tr>
<tr>
<td>KEY_RESIZE</td>
<td>Terminal resize event</td>
</tr>
<tr>
<td>KEY_MAX</td>
<td>Maximum key value</td>
</tr>
</tbody>
</table>
On VT100s and their software emulations, such as X terminal emulators, there are normally at least four function keys (KEY_F1, KEY_F2, KEY_F3, KEY_F4) available, and the arrow keys mapped to KEY_UP, KEY_DOWN, KEY_LEFT and KEY_RIGHT in the obvious way. If your machine has a PC keyboard, it is safe to expect arrow keys and twelve function keys (older PC keyboards may have only ten function keys); also, the following keypad mappings are standard:

<table>
<thead>
<tr>
<th>Keycap</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>KEY_IC</td>
</tr>
<tr>
<td>Delete</td>
<td>KEY_DC</td>
</tr>
<tr>
<td>Home</td>
<td>KEY_HOME</td>
</tr>
<tr>
<td>End</td>
<td>KEY_END</td>
</tr>
<tr>
<td>Page Up</td>
<td>KEY_PPAGE</td>
</tr>
<tr>
<td>Page Down</td>
<td>KEY_NPAGE</td>
</tr>
</tbody>
</table>

The following table lists characters from the alternate character set. These are inherited from the VT100 terminal, and will generally be available on software emulations such as X terminals. When there is no graphic available, curses falls back on a crude printable ASCII approximation.

**Note:** These are available only after `initscr()` has been called.

<table>
<thead>
<tr>
<th>ACS code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACS_BBSS</td>
<td>alternate name for upper right corner</td>
</tr>
<tr>
<td>ACS_BLOCK</td>
<td>solid square block</td>
</tr>
<tr>
<td>ACS_BOARD</td>
<td>board of squares</td>
</tr>
<tr>
<td>ACS_BSBS</td>
<td>alternate name for horizontal line</td>
</tr>
<tr>
<td>ACS_BSSB</td>
<td>alternate name for upper left corner</td>
</tr>
<tr>
<td>ACS_BSSS</td>
<td>alternate name for top tee</td>
</tr>
<tr>
<td>ACS_BTEE</td>
<td>bottom tee</td>
</tr>
<tr>
<td>ACS_BULLET</td>
<td>bullet</td>
</tr>
<tr>
<td>ACS_CKBOARD</td>
<td>checker board (stipple)</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>ACS_DARROW</td>
<td>arrow pointing down</td>
</tr>
<tr>
<td>ACS_DEGREE</td>
<td>degree symbol</td>
</tr>
<tr>
<td>ACS_DIAMOND</td>
<td>diamond</td>
</tr>
<tr>
<td>ACS_GEQUAL</td>
<td>greater-than-or-equal-to</td>
</tr>
<tr>
<td>ACS_HLINE</td>
<td>horizontal line</td>
</tr>
<tr>
<td>ACS_LANTERN</td>
<td>lantern symbol</td>
</tr>
<tr>
<td>ACS_LARROW</td>
<td>left arrow</td>
</tr>
<tr>
<td>ACS_LEQUAL</td>
<td>less-than-or-equal-to</td>
</tr>
<tr>
<td>ACS_LLCORNER</td>
<td>lower left-hand corner</td>
</tr>
<tr>
<td>ACS_LRCORNER</td>
<td>lower right-hand corner</td>
</tr>
<tr>
<td>ACS_LTEE</td>
<td>left tee</td>
</tr>
<tr>
<td>ACS_NEQUAL</td>
<td>not-equal sign</td>
</tr>
<tr>
<td>ACS_PI</td>
<td>letter pi</td>
</tr>
<tr>
<td>ACS_PLMINUS</td>
<td>plus-or-minus sign</td>
</tr>
<tr>
<td>ACS_PLUS</td>
<td>big plus sign</td>
</tr>
<tr>
<td>ACS_RARROW</td>
<td>right arrow</td>
</tr>
<tr>
<td>ACS_RTEE</td>
<td>right tee</td>
</tr>
<tr>
<td>ACS_S1</td>
<td>scan line 1</td>
</tr>
<tr>
<td>ACS_S3</td>
<td>scan line 3</td>
</tr>
<tr>
<td>ACS_S7</td>
<td>scan line 7</td>
</tr>
<tr>
<td>ACS_S9</td>
<td>scan line 9</td>
</tr>
<tr>
<td>ACS_SBBS</td>
<td>alternate name for lower right corner</td>
</tr>
<tr>
<td>ACS_SBSB</td>
<td>alternate name for vertical line</td>
</tr>
<tr>
<td>ACS_SBSS</td>
<td>alternate name for right tee</td>
</tr>
<tr>
<td>ACS_SSSB</td>
<td>alternate name for lower left corner</td>
</tr>
<tr>
<td>ACS_SSSBS</td>
<td>alternate name for bottom tee</td>
</tr>
<tr>
<td>ACS_SSSS</td>
<td>alternate name for left tee</td>
</tr>
<tr>
<td>ACS_SSSSS</td>
<td>alternate name for crossover or big plus</td>
</tr>
<tr>
<td>ACS_STERLING</td>
<td>pound sterling</td>
</tr>
<tr>
<td>ACS_TTEE</td>
<td>top tee</td>
</tr>
<tr>
<td>ACS_UARROW</td>
<td>up arrow</td>
</tr>
</tbody>
</table>
The following table lists the predefined colors:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR_BLACK</td>
<td>Black</td>
</tr>
<tr>
<td>COLOR_BLUE</td>
<td>Blue</td>
</tr>
<tr>
<td>COLOR_CYAN</td>
<td>Cyan (light greenish blue)</td>
</tr>
<tr>
<td>COLOR_GREEN</td>
<td>Green</td>
</tr>
<tr>
<td>COLOR_MAGENTA</td>
<td>Magenta (purplish red)</td>
</tr>
<tr>
<td>COLOR_RED</td>
<td>Red</td>
</tr>
<tr>
<td>COLOR_WHITE</td>
<td>White</td>
</tr>
<tr>
<td>COLOR_YELLOW</td>
<td>Yellow</td>
</tr>
</tbody>
</table>
The `curses.textpad` module provides a `Textbox` class that handles elementary text editing in a curses window, supporting a set of keybindings resembling those of Emacs (thus, also of Netscape Navigator, BBedit 6.x, FrameMaker, and many other programs). The module also provides a rectangle-drawing function useful for framing text boxes or for other purposes.

The module `curses.textpad` defines the following function:

```python
curses.textpad.rectangle(win, uly, ulx, lry, lrx)
```

Draw a rectangle. The first argument must be a window object; the remaining arguments are coordinates relative to that window. The second and third arguments are the y and x coordinates of the upper left hand corner of the rectangle to be drawn; the fourth and fifth arguments are the y and x coordinates of the lower right hand corner. The rectangle will be drawn using VT100/IBM PC forms characters on terminals that make this possible (including xterm and most other software terminal emulators). Otherwise it will be drawn with ASCII dashes, vertical bars, and plus signs.
15.12.1. Textbox objects

You can instantiate a `Textbox` object as follows:

```python
class curses.textpad.Textbox(win)
```

Return a textbox widget object. The `win` argument should be a `curses.WindowObject` in which the textbox is to be contained. The edit cursor of the textbox is initially located at the upper left hand corner of the containing window, with coordinates \((0, 0)\). The instance's `stripspaces` flag is initially on.

`Textbox` objects have the following methods:

```python
edit([validator])
```

This is the entry point you will normally use. It accepts editing keystrokes until one of the termination keystrokes is entered. If `validator` is supplied, it must be a function. It will be called for each keystroke entered with the keystroke as a parameter; command dispatch is done on the result. This method returns the window contents as a string; whether blanks in the window are included is affected by the `stripspaces` attribute.

```python
do_command(ch)
```

Process a single command keystroke. Here are the supported special keystrokes:

<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control-A</td>
<td>Go to left edge of window.</td>
</tr>
<tr>
<td>Control-B</td>
<td>Cursor left, wrapping to previous line if appropriate.</td>
</tr>
<tr>
<td>Control-D</td>
<td>Delete character under cursor.</td>
</tr>
<tr>
<td>Control-E</td>
<td>Go to right edge (stripspaces off) or end of line (stripspaces on).</td>
</tr>
</tbody>
</table>
Control-F | Cursor right, wrapping to next line when appropriate.
Control-G | Terminate, returning the window contents.
Control-H | Delete character backward.
Control-J | Terminate if the window is 1 line, otherwise insert newline.
Control-K | If line is blank, delete it, otherwise clear to end of line.
Control-L | Refresh screen.
Control-N | Cursor down; move down one line.
Control-O | Insert a blank line at cursor location.
Control-P | Cursor up; move up one line.

Move operations do nothing if the cursor is at an edge where the movement is not possible. The following synonyms are supported where possible:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Keystroke</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_LEFT</td>
<td>Control-B</td>
</tr>
<tr>
<td>KEY_RIGHT</td>
<td>Control-F</td>
</tr>
<tr>
<td>KEY_UP</td>
<td>Control-P</td>
</tr>
<tr>
<td>KEY_DOWN</td>
<td>Control-N</td>
</tr>
<tr>
<td>KEY_BACKSPACE</td>
<td>Control-h</td>
</tr>
</tbody>
</table>

All other keystrokes are treated as a command to insert the given character and move right (with line wrapping).

gather()

Return the window contents as a string; whether blanks in the window are included is affected by the stripspaces member.

stripspaces

This attribute is a flag which controls the interpretation of blanks in the window. When it is on, trailing blanks on each line are ignored; any cursor motion that would land the cursor on a trailing blank goes to the end of that line instead, and
trailing blanks are stripped when the window contents are gathered.
15.13. **curses.ascii** — Utilities for ASCII characters

*New in version 1.6.*

The *curses.ascii* module supplies name constants for ASCII characters and functions to test membership in various ASCII character classes. The constants supplied are names for control characters as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>Start of heading, console interrupt</td>
</tr>
<tr>
<td>SOH</td>
<td>Start of text</td>
</tr>
<tr>
<td>STX</td>
<td>End of text</td>
</tr>
<tr>
<td>ETX</td>
<td>End of transmission</td>
</tr>
<tr>
<td>EOT</td>
<td>Enquiry, goes with <em>ACK</em> flow control</td>
</tr>
<tr>
<td>ENQ</td>
<td>Acknowledgement</td>
</tr>
<tr>
<td>BEL</td>
<td>Bell</td>
</tr>
<tr>
<td>BS</td>
<td>Backspace</td>
</tr>
<tr>
<td>TAB</td>
<td>Tab</td>
</tr>
<tr>
<td>HT</td>
<td>Alias for <em>TAB</em>: “Horizontal tab”</td>
</tr>
<tr>
<td>LF</td>
<td>Line feed</td>
</tr>
<tr>
<td>NL</td>
<td>Alias for <em>LF</em>: “New line”</td>
</tr>
<tr>
<td>VT</td>
<td>Vertical tab</td>
</tr>
<tr>
<td>FF</td>
<td>Form feed</td>
</tr>
<tr>
<td>CR</td>
<td>Carriage return</td>
</tr>
<tr>
<td>SO</td>
<td>Shift-out, begin alternate character set</td>
</tr>
<tr>
<td>SI</td>
<td>Shift-in, resume default character set</td>
</tr>
<tr>
<td>DLE</td>
<td>Data-link escape</td>
</tr>
<tr>
<td>DC1</td>
<td>XON, for flow control</td>
</tr>
<tr>
<td>DC2</td>
<td>Device control 2, block-mode flow control</td>
</tr>
<tr>
<td>------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>DC3</td>
<td>XOFF, for flow control</td>
</tr>
<tr>
<td>DC4</td>
<td>Device control 4</td>
</tr>
<tr>
<td>NAK</td>
<td>Negative acknowledgement</td>
</tr>
<tr>
<td>SYN</td>
<td>Synchronous idle</td>
</tr>
<tr>
<td>ETB</td>
<td>End transmission block</td>
</tr>
<tr>
<td>CAN</td>
<td>Cancel</td>
</tr>
<tr>
<td>EM</td>
<td>End of medium</td>
</tr>
<tr>
<td>SUB</td>
<td>Substitute</td>
</tr>
<tr>
<td>ESC</td>
<td>Escape</td>
</tr>
<tr>
<td>FS</td>
<td>File separator</td>
</tr>
<tr>
<td>GS</td>
<td>Group separator</td>
</tr>
<tr>
<td>RS</td>
<td>Record separator, block-mode terminator</td>
</tr>
<tr>
<td>US</td>
<td>Unit separator</td>
</tr>
<tr>
<td>SP</td>
<td>Space</td>
</tr>
<tr>
<td>DEL</td>
<td>Delete</td>
</tr>
</tbody>
</table>

Note that many of these have little practical significance in modern usage. The mnemonics derive from teleprinter conventions that predate digital computers.

The module supplies the following functions, patterned on those in the standard C library:

```c
#include <curses.h>

int isalnum(int c) {
    return isalpha(c) || isdigit(c);
}
```

Checks for an ASCII alphanumeric character; it is equivalent to `isalpha(c)` or `isdigit(c)`.

```c
#include <curses.h>

int isalpha(int c) {
    return isupper(c) || islower(c);
}
```

Checks for an ASCII alphabetic character; it is equivalent to `isupper(c)` or `islower(c)`.

```c
#include <curses.h>

int isascii(int c) {
    return c >= 0 && c <= 127;
}
```

Checks for a valid ASCII character.
Checks for a character value that fits in the 7-bit ASCII set.

curses.ascii.isblank(c)
Checks for an ASCII whitespace character; space or horizontal tab.

curses.ascii.iscntrl(c)
Checks for an ASCII control character (in the range 0x00 to 0x1f or 0x7f).

curses.ascii.isdigit(c)
Checks for an ASCII decimal digit, '0' through '9'. This is equivalent to c in string.digits.

curses.ascii.isgraph(c)
Checks for ASCII any printable character except space.

curses.ascii.islower(c)
Checks for an ASCII lower-case character.

curses.ascii.isprint(c)
Checks for any ASCII printable character including space.

curses.ascii.ispunct(c)
Checks for any printable ASCII character which is not a space or an alphanumeric character.

curses.ascii.isspace(c)
Checks for ASCII white-space characters; space, line feed, carriage return, form feed, horizontal tab, vertical tab.

curses.ascii.isupper(c)
Checks for an ASCII uppercase letter.

curses.ascii.isxdigit(c)
Checks for an ASCII hexadecimal digit. This is equivalent to c in string.digits.
string.hexdigits.

curses.ascii.isctrl(c)
Checks for an ASCII control character (ordinal values 0 to 31).

curses.ascii.ismeta(c)
Checks for a non-ASCII character (ordinal values 0x80 and above).

These functions accept either integers or strings; when the argument is a string, it is first converted using the built-in function ord().

Note that all these functions check ordinal bit values derived from the first character of the string you pass in; they do not actually know anything about the host machine’s character encoding. For functions that know about the character encoding (and handle internationalization properly) see the string module.

The following two functions take either a single-character string or integer byte value; they return a value of the same type.

curses.ascii.ascii(c)
Return the ASCII value corresponding to the low 7 bits of c.

curses.ascii.ctrl(c)
Return the control character corresponding to the given character (the character bit value is bitwise-anded with 0x1f).

curses.ascii.alt(c)
Return the 8-bit character corresponding to the given ASCII character (the character bit value is bitwise-ored with 0x80).

The following function takes either a single-character string or integer value; it returns a string.

curses.ascii.unctrl(c)
Return a string representation of the ASCII character c. If c is printable, this string is the character itself. If the character is a control character (0x00–0x1f) the string consists of a caret ('^') followed by the corresponding uppercase letter. If the character is an ASCII delete (0x7f) the string is '^?'. If the character has its meta bit (0x80) set, the meta bit is stripped, the preceding rules applied, and '! ' prepended to the result.

curses.ascii.controlnames

A 33-element string array that contains the ASCII mnemonics for the thirty-two ASCII control characters from 0 (NUL) to 0x1f (US), in order, plus the mnemonic sp for the space character.
Panels are windows with the added feature of depth, so they can be stacked on top of each other, and only the visible portions of each window will be displayed. Panels can be added, moved up or down in the stack, and removed.
The module `curses.panel` defines the following functions:

- `curses.panel.bottom_panel()`
  Returns the bottom panel in the panel stack.

- `curses.panel.new_panel(win)`
  Returns a panel object, associating it with the given window `win`. Be aware that you need to keep the returned panel object referenced explicitly. If you don’t, the panel object is garbage collected and removed from the panel stack.

- `curses.panel.top_panel()`
  Returns the top panel in the panel stack.

- `curses.panel.update_panels()`
  Updates the virtual screen after changes in the panel stack. This does not call `curses.doupdate()`, so you'll have to do this yourself.
15.14.2. Panel Objects

Panel objects, as returned by \texttt{new\_panel()} above, are windows with a stacking order. There's always a window associated with a panel which determines the content, while the panel methods are responsible for the window's depth in the panel stack.

Panel objects have the following methods:

\begin{itemize}
  \item \texttt{Panel\_above()} \\
      Returns the panel above the current panel.
  \item \texttt{Panel\_below()} \\
      Returns the panel below the current panel.
  \item \texttt{Panel\_bottom()} \\
      Push the panel to the bottom of the stack.
  \item \texttt{Panel\_hidden()} \\
      Returns true if the panel is hidden (not visible), false otherwise.
  \item \texttt{Panel\_hide()} \\
      Hide the panel. This does not delete the object, it just makes the window on screen invisible.
  \item \texttt{Panel\_move(y, x)} \\
      Move the panel to the screen coordinates \((y, x)\).
  \item \texttt{Panel\_replace(win)} \\
      Change the window associated with the panel to the window \texttt{win}.
  \item \texttt{Panel\_set\_userptr(obj)} \\
      Set the panel's user pointer to \texttt{obj}. This is used to associate an arbitrary piece of data with the panel, and can be any Python
Panel. **show()**
Display the panel (which might have been hidden).

Panel. **top()**
Push panel to the top of the stack.

Panel. **userptr()**
Returns the user pointer for the panel. This might be any Python object.

Panel. **window()**
Returns the window object associated with the panel.
15.15. platform — Access to underlying platform’s identifying data

*New in version 2.3.*

**Source code:** Lib/platform.py

---

**Note:** Specific platforms listed alphabetically, with Linux included in the Unix section.
15.15.1. Cross Platform

platform.architecture(executable=sys.executable, bits='', linkage='')

Queries the given executable (defaults to the Python interpreter binary) for various architecture information.

Returns a tuple (bits, linkage) which contain information about the bit architecture and the linkage format used for the executable. Both values are returned as strings.

Values that cannot be determined are returned as given by the parameter presets. If bits is given as '', the sizeof(pointer)() (or sizeof(long)() on Python version < 1.5.2) is used as indicator for the supported pointer size.

The function relies on the system's file command to do the actual work. This is available on most if not all Unix platforms and some non-Unix platforms and then only if the executable points to the Python interpreter. Reasonable defaults are used when the above needs are not met.

**Note:** On Mac OS X (and perhaps other platforms), executable files may be universal files containing multiple architectures.

To get at the “64-bitness” of the current interpreter, it is more reliable to query the sys.maxsize attribute:

```python
is_64bits = sys.maxsize > 2**32
```

platform.machine()

Returns the machine type, e.g. 'i386'. An empty string is returned if the value cannot be determined.
platform.\texttt{node}()  
Returns the computer’s network name (may not be fully qualified!). An empty string is returned if the value cannot be determined.

platform.\texttt{platform}(\texttt{aliased}=0, \texttt{terse}=0)  
Returns a single string identifying the underlying platform with as much useful information as possible.

The output is intended to be \textit{human readable} rather than machine parseable. It may look different on different platforms and this is intended.

If \texttt{aliased} is true, the function will use aliases for various platforms that report system names which differ from their common names, for example SunOS will be reported as Solaris. The \texttt{system\_alias()} function is used to implement this.

Setting \texttt{terse} to true causes the function to return only the absolute minimum information needed to identify the platform.

platform.\texttt{processor}()  
Returns the (real) processor name, e.g. `\texttt{amdk6}`.

An empty string is returned if the value cannot be determined. Note that many platforms do not provide this information or simply return the same value as for \texttt{machine()}. NetBSD does this.

platform.\texttt{python\_build}()  
Returns a tuple \texttt{(buildno, builddate)} stating the Python build number and date as strings.

platform.\texttt{python\_compiler}()  
Returns a string identifying the compiler used for compiling Python.
platform. **python_branch**()

Returns a string identifying the Python implementation SCM branch.

*New in version 2.6.*

platform. **python_implementation**()


*New in version 2.6.*

platform. **python_revision**()

Returns a string identifying the Python implementation SCM revision.

*New in version 2.6.*

platform. **python_version**()

Returns the Python version as string `'major.minor.patchlevel'`.

Note that unlike the Python `sys.version`, the returned value will always include the patchlevel (it defaults to 0).

platform. **python_version_tuple**()

Returns the Python version as tuple `(major, minor, patchlevel)` of strings.

Note that unlike the Python `sys.version`, the returned value will always include the patchlevel (it defaults to `'0'`).

platform. **release**()

Returns the system’s release, e.g. `'2.2.0'` or `'NT'`. An empty string is returned if the value cannot be determined.

platform. **system**()
Returns the system/OS name, e.g. 'Linux', 'Windows', or 'Java'. An empty string is returned if the value cannot be determined.

platform.system_alias(system, release, version)

Returns (system, release, version) aliased to common marketing names used for some systems. It also does some reordering of the information in some cases where it would otherwise cause confusion.

platform.version()

Returns the system's release version, e.g. '#3 on degas'. An empty string is returned if the value cannot be determined.

platform.uname()

Fairly portable uname interface. Returns a tuple of strings (system, node, release, version, machine, processor) identifying the underlying platform.

Note that unlike the os.uname() function this also returns possible processor information as additional tuple entry.

Entries which cannot be determined are set to ''.
15.15.2. Java Platform

platform.java_ver(release='', vendor='', vminfo=('', '', ''), osinfo=(''))

Version interface for Jython.

Returns a tuple (release, vendor, vminfo, osinfo) with vminfo being a tuple (vm_name, vm_release, vm_vendor) and osinfo being a tuple (os_name, os_version, os_arch). Values which cannot be determined are set to the defaults given as parameters (which all default to '').
15.15.3. Windows Platform

platform.win32_ver(release='', version='', csd='', ptype='')

Get additional version information from the Windows Registry and return a tuple (release, version, csd, ptype) referring to OS release, version number, CSD level (service pack) and OS type (multi/single processor).

As a hint: *ptype* is 'Uniprocessor Free' on single processor NT machines and 'Multiprocessor Free' on multi processor machines. The 'Free' refers to the OS version being free of debugging code. It could also state 'Checked' which means the OS version uses debugging code, i.e. code that checks arguments, ranges, etc.

*Note:* This function works best with Mark Hammond's *win32all* package installed, but also on Python 2.3 and later (support for this was added in Python 2.6). It obviously only runs on Win32 compatible platforms.

15.15.3.1. Win95/98 specific

platform.popen(cmd, mode='r', bufsize=None)

Portable *popen()* interface. Find a working popen implementation preferring *win32pipe.popen()*. On Windows NT, *win32pipe.popen()* should work; on Windows 9x it hangs due to bugs in the MS C library.
15.15.4. Mac OS Platform

platform.mac_ver(release='', versioninfo=('','',''), machine='')

Get Mac OS version information and return it as tuple (release, versioninfo, machine) with versioninfo being a tuple (version, dev_stage, non_release_version).

Entries which cannot be determined are set to '' . All tuple entries are strings.
### 15.15.5. Unix Platforms

```python
platform.dist(distname='', version='', id='', supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...))
```

This is an old version of the functionality now provided by `linux_distribution()`. For new code, please use the `linux_distribution()`.

The only difference between the two is that `dist()` always returns the short name of the distribution taken from the `supported_dists` parameter.

**Deprecated since version 2.6.**

```python
platform.linux_distribution(distname='', version='', id='', supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...), full_distribution_name=1)
```

Tries to determine the name of the Linux OS distribution name.

`supported_dists` may be given to define the set of Linux distributions to look for. It defaults to a list of currently supported Linux distributions identified by their release file name.

If `full_distribution_name` is true (default), the full distribution read from the OS is returned. Otherwise the short name taken from `supported_dists` is used.

Returns a tuple `(distname,version,id)` which defaults to the args given as parameters. `id` is the item in parentheses after the version number. It is usually the version codename.

*New in version 2.6.*
platform.libc_ver(executable=sys.executable, lib='', version='',
chunksize=2048)

Tries to determine the libc version against which the file executable (defaults to the Python interpreter) is linked. Returns a tuple of strings (lib, version) which default to the given parameters in case the lookup fails.

Note that this function has intimate knowledge of how different libc versions add symbols to the executable is probably only usable for executables compiled using gcc.

The file is read and scanned in chunks of chunksize bytes.
15. Generic Operating System Services
15.16. **errno** — Standard errno system symbols

This module makes available standard `errno` system symbols. The value of each symbol is the corresponding integer value. The names and descriptions are borrowed from `linux/include/errno.h`, which should be pretty all-inclusive.

```python
errno.errorcode
```

Dictionary providing a mapping from the errno value to the string name in the underlying system. For instance, `errno.errorcode[errno.EPERM]` maps to 'EPERM'.

To translate a numeric error code to an error message, use `os.strerror()`.

Of the following list, symbols that are not used on the current platform are not defined by the module. The specific list of defined symbols is available as `errno.errorcode.keys()`. Symbols available can include:

```python
errno.EPERM
```

Operation not permitted

```python
errno.ENOENT
```

No such file or directory

```python
errno.ESRCH
```

No such process

```python
errno.EINTR
```

Interrupted system call

```python
errno.EIO
```
I/O error

errno. ENXIO
    No such device or address

errno. E2BIG
    Arg list too long

errno. ENOEXEC
    Exec format error

errno. EBADF
    Bad file number

errno. ECHILD
    No child processes

errno. EAGAIN
    Try again

errno. ENOMEM
    Out of memory

errno. EACCES
    Permission denied

errno.EFAULT
    Bad address

errno. ENOTBLK
    Block device required

errno. EBUSY
    Device or resource busy

errno. EEXIST
    File exists
**errno.** **EXDEV**
Cross-device link

**errno.** **ENODEV**
No such device

**errno.** **ENOTDIR**
Not a directory

**errno.** **EISDIR**
Is a directory

**errno.** **EINVAL**
Invalid argument

**errno.** **ENFILE**
File table overflow

**errno.** **EMFILE**
Too many open files

**errno.** **ENOTTY**
Not a typewriter

**errno.** **ETXTBSY**
Text file busy

**errno.** **EFBIG**
File too large

**errno.** **ENOSPC**
No space left on device

**errno.** **ESPIPE**
Illegal seek

**errno.** **EROFS**
Read-only file system
errno. **EMLINK**
Too many links

errno. **EPIPE**
Broken pipe

errno. **EDOM**
Math argument out of domain of func

errno. **ERANGE**
Math result not representable

errno. **EDEADLK**
Resource deadlock would occur

errno. **ENAMETOOLONG**
File name too long

errno. **ENOLCK**
No record locks available

errno. **ENOSYS**
Function not implemented

errno. **ENOTEMPTY**
Directory not empty

errno. **ELOOP**
Too many symbolic links encountered

errno. **EWOULDBLOCK**
Operation would block

errno. **ENOMSG**
No message of desired type

errno. **EIDRM**
Identifier removed
errno. **ECHRRNG**
Channel number out of range

errno. **EL2NSYNC**
Level 2 not synchronized

errno. **EL3HLT**
Level 3 halted

errno. **EL3RST**
Level 3 reset

errno. **ELNRNG**
Link number out of range

errno. **EUNATCH**
Protocol driver not attached

errno. **ENOCSI**
No CSI structure available

errno. **EL2HLT**
Level 2 halted

errno. **EBADE**
Invalid exchange

errno. **EBADR**
Invalid request descriptor

errno. **EXFULL**
Exchange full

errno. **ENOANO**
No anode

errno. **EBADRQC**
Invalid request code
errno. **EBADSLT**
Invalid slot

errno. **EDEADLOCK**
File locking deadlock error

errno. **EBFONT**
Bad font file format

errno. **ENOSTR**
Device not a stream

errno. **ENODATA**
No data available

errno. **ETIME**
Timer expired

errno. **ENOSR**
Out of streams resources

errno. **ENONET**
Machine is not on the network

errno. **ENOPKG**
Package not installed

errno. **EREMOTE**
Object is remote

errno. **ENOLINK**
Link has been severed

errno. **EADV**
Advertise error

errno. **ESRMNT**
Srmount error
errno. **ECOMM**
Communication error on send

errno. **EPROTO**
Protocol error

errno. **EMULTIHOP**
Multihop attempted

errno. **EDOTDOT**
RFS specific error

errno. **EBADMSG**
Not a data message

errno. **EOVERFLOW**
Value too large for defined data type

errno. **ENOTUNIQ**
Name not unique on network

errno. **EBADFD**
File descriptor in bad state

errno. **EREMCHG**
Remote address changed

errno. **ELIBACC**
Can not access a needed shared library

errno. **ELIBBAD**
Accessing a corrupted shared library

errno. **ELIBSCN**
.lib section in a.out corrupted

errno. **ELIBMAX**
Attempting to link in too many shared libraries
errno. **ELIBEXEC**
    Cannot exec a shared library directly

errno. **EILSEQ**
    Illegal byte sequence

errno. **ERESTART**
    Interrupted system call should be restarted

errno. **ESTRPIPE**
    Streams pipe error

errno. **EUSERS**
    Too many users

errno. **ENOTSOCK**
    Socket operation on non-socket

errno. **EDESTADDRREQ**
    Destination address required

errno. **EMSGSIZE**
    Message too long

errno. **EPROTOTYPE**
    Protocol wrong type for socket

errno. **ENOPROTOOPT**
    Protocol not available

errno. **EPROTONOSUPPORT**
    Protocol not supported

errno. **ESOCKTNOSUPPORT**
    Socket type not supported

errno. **EOPNOTSUPP**
    Operation not supported on transport endpoint
errno. **EPFNOSUPPORT**
   Protocol family not supported

errno. **EAFNOSUPPORT**
   Address family not supported by protocol

errno. **EADDRINUSE**
   Address already in use

errno. **EADDRNOTAVAIL**
   Cannot assign requested address

errno. **ENETDOWN**
   Network is down

errno. **ENETUNREACH**
   Network is unreachable

errno. **ENETRESET**
   Network dropped connection because of reset

errno. **ECONNABORTED**
   Software caused connection abort

errno. **ECONNRESET**
   Connection reset by peer

errno. **ENOBUFFS**
   No buffer space available

errno. **EISCONN**
   Transport endpoint is already connected

errno. **ENOTCONN**
   Transport endpoint is not connected

errno. **ESHUTDOWN**
   Cannot send after transport endpoint shutdown
errno. **ETOOMANYREFS**
Too many references: cannot splice

errno. **ETIMEDOUT**
Connection timed out

errno. **ECONNREFUSED**
Connection refused

errno. **EHOSTDOWN**
Host is down

errno. **EHOSTUNREACH**
No route to host

errno. **EALREADY**
Operation already in progress

errno. **EINPROGRESS**
Operation now in progress

errno. **ESTALE**
Stale NFS file handle

errno. **EUCLEAN**
Structure needs cleaning

errno. **ENOTNAM**
Not a XENIX named type file

errno. **ENAVAIL**
No XENIX semaphores available

errno. **EISNAM**
Is a named type file

errno. **EREMOTEIO**
Remote I/O error
errno.**EDQUOT**

Quota exceeded
15.17. `ctypes` — A foreign function library for Python

*New in version 2.5.*

`ctypes` is a foreign function library for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python.
15.17.1. ctypes tutorial

Note: The code samples in this tutorial use doctest to make sure that they actually work. Since some code samples behave differently under Linux, Windows, or Mac OS X, they contain doctest directives in comments.

Note: Some code samples reference the ctypes c_int type. This type is an alias for the c_long type on 32-bit systems. So, you should not be confused if c_long is printed if you would expect c_int — they are actually the same type.

15.17.1.1. Loading dynamic link libraries

ctypes exports the cdll, and on Windows windll and oledll objects, for loading dynamic link libraries.

You load libraries by accessing them as attributes of these objects. cdll loads libraries which export functions using the standard cdecl calling convention, while windll libraries call functions using the stdcall calling convention. oledll also uses the stdcall calling convention, and assumes the functions return a Windows HRESULT error code. The error code is used to automatically raise a WindowsError exception when the function call fails.

Here are some examples for Windows. Note that msvcr is the MS standard C library containing most standard C functions, and uses the cdecl calling convention:

```python
>>> from ctypes import *
>>> print windll.kernel32
<WinDLL 'kernel32', handle ... at ...>
>>> print cdll.msvcr
<CDLL 'msvcr', handle ... at ...>
```
Windows appends the usual `.dll` file suffix automatically.

On Linux, it is required to specify the filename including the extension to load a library, so attribute access can not be used to load libraries. Either the `LoadLibrary()` method of the dll loaders should be used, or you should load the library by creating an instance of CDLL by calling the constructor:

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
```

15.17.1.2. Accessing functions from loaded dlls

Functions are accessed as attributes of dll objects:

```
>>> from ctypes import *
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print windll.kernel32.GetModuleHandleA
<_FuncPtr object at 0x...>
>>> print windll.kernel32.MyOwnFunction
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
```

Note that win32 system dlls like `kernel32` and `user32` often export ANSI as well as UNICODE versions of a function. The UNICODE version is exported with an `w` appended to the name, while the ANSI version is exported with an `A` appended to the name. The win32
GetModuleHandle function, which returns a *module handle* for a given module name, has the following C prototype, and a macro is used to expose one of them as GetModuleHandle depending on whether UNICODE is defined or not:

```c
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

*windll* does not try to select one of them by magic, you must access the version you need by specifying `GetModuleHandleA` or `GetModuleHandleW` explicitly, and then call it with strings or unicode strings respectively.

Sometimes, dlls export functions with names which aren’t valid Python identifiers, like "??2@YAPAXI@Z". In this case you have to use `getattr()` to retrieve the function:

```python
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
```

On Windows, some dlls export functions not by name but by ordinal. These functions can be accessed by indexing the dll object with the ordinal number:

```python
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "ctypes.py", line 310, in __getitem__
    func = __stdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
```

15.17.1.3. Calling functions
You can call these functions like any other Python callable. This example uses the `time()` function, which returns system time in seconds since the Unix epoch, and the `GetModuleHandleA()` function, which returns a win32 module handle.

This example calls both functions with a NULL pointer (`None` should be used as the NULL pointer):

```python
>>> print libc.time(None)
1150640792
>>> print hex(windll.kernel32.GetModuleHandleA(None))
0x1d000000
```

c-types tries to protect you from calling functions with the wrong number of arguments or the wrong calling convention. Unfortunately this only works on Windows. It does this by examining the stack after the function returns, so although an error is raised the function has been called:

```python
>>> windll.kernel32.GetModuleHandleA()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Procedure probably called with not enough arguments
```

```python
>>> windll.kernel32.GetModuleHandleA(0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Procedure probably called with too many arguments
```

The same exception is raised when you call an stdcall function with the cdecl calling convention, or vice versa:

```python
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Procedure probably called with not enough arguments
```
To find out the correct calling convention you have to look into the C header file or the documentation for the function you want to call.

On Windows, `ctypes` uses win32 structured exception handling to prevent crashes from general protection faults when functions are called with invalid argument values:

```python
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
WindowsError: exception: access violation reading 0x00000020
```

There are, however, enough ways to crash Python with `ctypes`, so you should be careful anyway.

`None`, integers, longs, byte strings and unicode strings are the only native Python objects that can directly be used as parameters in these function calls. `None` is passed as a C NULL pointer, byte strings and unicode strings are passed as pointer to the memory block that contains their data (`char *` or `wchar_t *`). Python integers and Python longs are passed as the platforms default C `int` type, their value is masked to fit into the C type.

Before we move on calling functions with other parameter types, we have to learn more about `ctypes` data types.

15.17.1.4. Fundamental data types
**ctypes** defines a number of primitive C compatible data types:

<table>
<thead>
<tr>
<th>ctypes type</th>
<th>C type</th>
<th>Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_bool</td>
<td>_Bool</td>
<td>bool (1)</td>
</tr>
<tr>
<td>c_char</td>
<td>char</td>
<td>1-character string</td>
</tr>
<tr>
<td>c_wchar</td>
<td>wchar_t</td>
<td>1-character unicode string</td>
</tr>
<tr>
<td>c_byte</td>
<td>char</td>
<td>int/long</td>
</tr>
<tr>
<td>c_ubyte</td>
<td>unsigned char</td>
<td>int/long</td>
</tr>
<tr>
<td>c_short</td>
<td>short</td>
<td>int/long</td>
</tr>
<tr>
<td>c_ushort</td>
<td>unsigned short</td>
<td>int/long</td>
</tr>
<tr>
<td>c_int</td>
<td>int</td>
<td>int/long</td>
</tr>
<tr>
<td>c_uint</td>
<td>unsigned int</td>
<td>int/long</td>
</tr>
<tr>
<td>c_long</td>
<td>long</td>
<td>int/long</td>
</tr>
<tr>
<td>c_ulong</td>
<td>unsigned long</td>
<td>int/long</td>
</tr>
<tr>
<td>c_longlong</td>
<td>__int64 Or long long</td>
<td>int/long</td>
</tr>
<tr>
<td>c_ulonglong</td>
<td>unsigned __int64 Or unsigned long long</td>
<td>int/long</td>
</tr>
<tr>
<td>c_float</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>c_double</td>
<td>double</td>
<td>float</td>
</tr>
<tr>
<td>c_longdouble</td>
<td>long double</td>
<td>float</td>
</tr>
<tr>
<td>c_char_p</td>
<td>char * (NUL terminated)</td>
<td>string or None</td>
</tr>
<tr>
<td>c_wchar_p</td>
<td>wchar_t * (NUL terminated)</td>
<td>unicode or None</td>
</tr>
<tr>
<td>c_void_p</td>
<td>void *</td>
<td>int/long or None</td>
</tr>
</tbody>
</table>

1. The constructor accepts any object with a truth value.

All these types can be created by calling them with an optional initializer of the correct type and value:

```python
>>> c_int()
c_long(0)
```
```python
>>> c_char_p("Hello, World")
c_char_p('Hello, World')
```
```python
>>> c_ushort(-3)
```
Since these types are mutable, their value can also be changed afterwards:

```python
>>> i = c_int(42)
>>> print i
>>> c_long(42)
>>> print i.value
42
>>> i.value = -99
>>> print i.value
-99
```  
Assigning a new value to instances of the pointer types `c_char_p`, `c_wchar_p`, and `c_void_p` changes the memory location they point to, not the contents of the memory block (of course not, because Python strings are immutable):

```python
>>> s = "Hello, World"
>>> c_s = c_char_p(s)
>>> print c_s
>>> c_char_p('Hello, World')
>>> c_s.value = "Hi, there"
>>> print c_s
>>> c_char_p('Hi, there')
>>> print s
```

You should be careful, however, not to pass them to functions expecting pointers to mutable memory. If you need mutable memory blocks, ctypes has a `create_string_buffer()` function which creates these in various ways. The current memory block contents can be accessed (or changed) with the `raw` property; if you want to access it as NUL terminated string, use the `value` property:
The `create_string_buffer()` function replaces the `c_buffer()` function (which is still available as an alias), as well as the `c_string()` function from earlier ctypes releases. To create a mutable memory block containing unicode characters of the C type `wchar_t` use the `create_unicode_buffer()` function.

15.17.1.5. Calling functions, continued

Note that printf prints to the real standard output channel, not to `sys.stdout`, so these examples will only work at the console prompt, not from within IDLE or PythonWin:
As has been mentioned before, all Python types except integers, strings, and unicode strings have to be wrapped in their corresponding ctypes type, so that they can be converted to the required C data type:

```python
>>> printf("An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>> 
```

## 15.17.1.6. Calling functions with your own custom data types

You can also customize ctypes argument conversion to allow instances of your own classes be used as function arguments. ctypes looks for an _as_parameter_ attribute and uses this as the function argument. Of course, it must be one of integer, string, or unicode:

```python
>>> class Bottles(object):
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf("%d bottles of beer\n", bottles)
42 bottles of beer
19
>>> 
```

If you don’t want to store the instance’s data in the _as_parameter_ instance variable, you could define a property() which makes the data available.
15.17.1.7. Specifying the required argument types (function prototypes)

It is possible to specify the required argument types of functions exported from DLLs by setting the `argtypes` attribute.

`argtypes` must be a sequence of C data types (the `printf` function is probably not a good example here, because it takes a variable number and different types of parameters depending on the format string, on the other hand this is quite handy to experiment with this feature):

```python
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf("String '%s', Int %d, Double %f\n", "Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>> 
```

Specifying a format protects against incompatible argument types (just as a prototype for a C function), and tries to convert the arguments to valid types:

```python
>>> printf("%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ArgumentError: argument 2: exceptions.TypeError: wrong type
>>> printf("%s %d %f\n", "X", 2, 3)
X 2 3.000000
13
>>> 
```

If you have defined your own classes which you pass to function calls, you have to implement a `from_param()` class method for them to be able to use them in the `argtypes` sequence. The `from_param()` class method receives the Python object passed to the function call, it should do a typecheck or whatever is needed to make sure this object is acceptable, and then return the object itself, its
_as_parameter_ attribute, or whatever you want to pass as the C function argument in this case. Again, the result should be an integer, string, unicode, a ctypes instance, or an object with an _as_parameter_ attribute.

### 15.17.1.8. Return types

By default functions are assumed to return the C int type. Other return types can be specified by setting the restype attribute of the function object.

Here is a more advanced example, it uses the `strchr` function, which expects a string pointer and a char, and returns a pointer to a string:

```python
>>> strchr = libc.strchr
>>> strchr("abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p  # c_char_p is a pointer to a string
>>> strchr("abcdef", ord("d"))
'def'
>>> print strchr("abcdef", ord("x"))
None
```

If you want to avoid the `ord("x")` calls above, you can set the argtypes attribute, and the second argument will be converted from a single character Python string into a C char:

```python
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr("abcdef", "d")
'def'
>>> strchr("abcdef", "def")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ArgumentError: argument 2: exceptions.TypeError: one character
>>> print strchr("abcdef", "x")
None
```
You can also use a callable Python object (a function or a class for example) as the `restype` attribute, if the foreign function returns an integer. The callable will be called with the `integer` the C function returns, and the result of this call will be used as the result of your function call. This is useful to check for error return values and automatically raise an exception:

```python
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
... >>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in ValidHandle
WindowsError: [Errno 126] The specified module could not be found.
```

`WinError` is a function which will call Windows `FormatMessage()` api to get the string representation of an error code, and `returns` an exception. `WinError` takes an optional error code parameter, if no one is used, it calls `GetLastError()` to retrieve it.

Please note that a much more powerful error checking mechanism is available through the `errcheck` attribute; see the reference manual for details.

15.17.1.9. Passing pointers (or: passing
parameters by reference)

Sometimes a C api function expects a `pointer` to a data type as parameter, probably to write into the corresponding location, or if the data is too large to be passed by value. This is also known as *passing parameters by reference*.

`ctypes` exports the `byref()` function which is used to pass parameters by reference. The same effect can be achieved with the `pointer()` function, although `pointer()` does a lot more work since it constructs a real pointer object, so it is faster to use `byref()` if you don't need the pointer object in Python itself:

```python
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(\000 * 32)
>>> print i.value, f.value, repr(s.value)
0 0.0 '
>>> libc sscanf("1 3.14 Hello", "%d %f %s",
... byref(i), byref(f), s)
3
>>> print i.value, f.value, repr(s.value)
1 3.1400001049 'Hello'
```  

15.17.1.10. Structures and unions

Structures and unions must derive from the `Structure` and `Union` base classes which are defined in the `ctypes` module. Each subclass must define a `_fields_` attribute. `_fields_` must be a list of 2-tuples, containing a field name and a field type.

The field type must be a `ctypes` type like `c_int`, or any other derived `ctypes` type: structure, union, array, pointer.

Here is a simple example of a POINT structure, which contains two
integers named $x$ and $y$, and also shows how to initialize a structure in the constructor:

```python
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
... >>> point = POINT(10, 20)
>>> print point.x, point.y
10 20
>>> point = POINT(y=5)
>>> print point.x, point.y
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: too many initializers

You can, however, build much more complicated structures. A structure can itself contain other structures by using a structure as a field type.

Here is a RECT structure which contains two POINTs named *upperleft* and *lowerright*:

```python
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
... >>> rc = RECT(point)
>>> print rc.upperleft.x, rc.upperleft.y
0 5
>>> print rc.lowerright.x, rc.lowerright.y
0 0

Nested structures can also be initialized in the constructor in several ways:

```python
>>> r = RECT(POINT(1, 2), POINT(3, 4))
```
```python
>>> r = RECT((1, 2), (3, 4))
```

Field descriptors can be retrieved from the class, they are useful for debugging because they can provide useful information:

```python
>>> print POINT.x
<Field type=c_long, ofs=0, size=4>
>>> print POINT.y
<Field type=c_long, ofs=4, size=4>
```

**Warning:** `ctypes` does not support passing unions or structures with bit-fields to functions by value. While this may work on 32-bit x86, it’s not guaranteed by the library to work in the general case. Unions and structures with bit-fields should always be passed to functions by pointer.

### 15.17.1.11. Structure/union alignment and byte order

By default, Structure and Union fields are aligned in the same way the C compiler does it. It is possible to override this behavior by specifying a `_pack_` class attribute in the subclass definition. This must be set to a positive integer and specifies the maximum alignment for the fields. This is what `#pragma pack(n)` also does in MSVC.

`ctypes` uses the native byte order for Structures and Unions. To build structures with non-native byte order, you can use one of the `BigEndianStructure`, `LittleEndianStructure`, `BigEndianUnion`, and `LittleEndianUnion` base classes. These classes cannot contain pointer fields.

### 15.17.1.12. Bit fields in structures and unions
It is possible to create structures and unions containing bit fields. Bit fields are only possible for integer fields, the bit width is specified as the third item in the _fields_ tuples:

```python
>>> class Int(Structure):
...     _fields_ = ["first_16", c_int, 16],
...             ("second_16", c_int, 16]
... >>> print(Int.first_16
<Field type=c_long, ofs=0:0, bits=16>
>>> print(Int.second_16
<Field type=c_long, ofs=0:16, bits=16>
```

15.17.1.13. Arrays

Arrays are sequences, containing a fixed number of instances of the same type.

The recommended way to create array types is by multiplying a data type with a positive integer:

```python
TenPointsArrayType = POINT * 10
```

Here is an example of a somewhat artificial data type, a structure containing 4 POINTs among other stuff:

```python
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
... >>>
>>> class MyStruct(Structure):
...     _fields_ = ("a", c_int),
...             ("b", c_float),
...             ("point_array", POINT * 4]
... >>>
>>> print(len(MyStruct().point_array))
4
```
Instances are created in the usual way, by calling the class:

```python
arr = TenPointsArrayType()
for pt in arr:
    print pt.x, pt.y
```

The above code print a series of 0 0 lines, because the array contents is initialized to zeros.

Initializers of the correct type can also be specified:

```python
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print ii
<c_long_Array_10 object at 0x...>
>>> for i in ii:
...     print i,
1 2 3 4 5 6 7 8 9 10

15.17.1.14. Pointers

Pointer instances are created by calling the `pointer()` function on a `ctypes` type:

```python
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
```

Pointer instances have a `contents` attribute which returns the object to which the pointer points, the `i` object above:

```python
>>> pi.contents
<types.c_long object at 0x...>
>>> c_long(42)
```
Note that `ctypes` does not have OOR (original object return), it constructs a new, equivalent object each time you retrieve an attribute:

```python
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
```  

Assigning another `c_int` instance to the pointer’s contents attribute would cause the pointer to point to the memory location where this is stored:

```python
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
99
```  

Pointer instances can also be indexed with integers:

```python
>>> pi[0]
99
```  

Assigning to an integer index changes the pointed to value:

```python
>>> print i
99
>>> pi[0] = 22
>>> print i
22
```  

It is also possible to use indexes different from 0, but you must know what you’re doing, just as in C: You can access or change arbitrary memory locations. Generally you only use this feature if you receive a pointer from a C function, and you *know* that the pointer actually
points to an array instead of a single item.

Behind the scenes, the `pointer()` function does more than simply create pointer instances, it has to create pointer types first. This is done with the `POINTER()` function, which accepts any `ctypes` type, and returns a new type:

```python
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes._Pointer'>
>>> PI(42)
<ctypes._Pointer object at 0x...>
```

Calling the pointer type without an argument creates a `NULL` pointer. `NULL` pointers have a `False` boolean value:

```python
>>> null_ptr = POINTER(c_int)()
>>> print bool(null_ptr)
False
```

`ctypes` checks for `NULL` when dereferencing pointers (but dereferencing invalid non-`NULL` pointers would crash Python):

```python
>>> null_ptr[0]
Traceback (most recent call last):
  ....
ValueError: NULL pointer access
```

```python
>>> null_ptr[0] = 1234
Traceback (most recent call last):
  ....
ValueError: NULL pointer access
```
15.17.1.15. Type conversions

Usually, ctypes does strict type checking. This means, if you have \texttt{POINTER(c\_int)} in the \texttt{argtypes} list of a function or as the type of a member field in a structure definition, only instances of exactly the same type are accepted. There are some exceptions to this rule, where ctypes accepts other objects. For example, you can pass compatible array instances instead of pointer types. So, for \texttt{POINTER(c\_int)}, ctypes accepts an array of \texttt{c\_int}:

\begin{verbatim}
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))
... >>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print bar.values[i]
... 1
2
3
>>> bar.values = None

>>> for i in range(bar.count):
...     print bar.values[i]
...

In addition, if a function argument is explicitly declared to be a pointer type (such as \texttt{POINTER(c\_int)}) in \texttt{argtypes}, an object of the pointed type (\texttt{c\_int} in this case) can be passed to the function. ctypes will apply the required \texttt{byref()} conversion in this case automatically.

To set a \texttt{POINTER} type field to \texttt{NULL}, you can assign \texttt{None}:

\begin{verbatim}
>>> bar.values = None

>>> for i in range(bar.count):
...     print bar.values[i]
...

Sometimes you have instances of incompatible types. In C, you can cast one type into another type. ctypes provides a \texttt{cast()} function
which can be used in the same way. The `Bar` structure defined above accepts `POINTER(c_int)` pointers or `c_int` arrays for its `values` field, but not instances of other types:

```
>>> bar.values = (c_byte * 4)()
TypeError: incompatible types, c_byte_Array_4 instance instead
```

For these cases, the `cast()` function is handy.

The `cast()` function can be used to cast a ctypes instance into a pointer to a different ctypes data type. `cast()` takes two parameters, a ctypes object that is or can be converted to a pointer of some kind, and a ctypes pointer type. It returns an instance of the second argument, which references the same memory block as the first argument:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
```

So, `cast()` can be used to assign to the `values` field of `Bar` the structure:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print bar.values[0]
0
```

15.17.1.16. Incomplete Types

`Incomplete Types` are structures, unions or arrays whose members are not yet specified. In C, they are specified by forward
declarations, which are defined later:

```c
struct cell; /* forward declaration */
struct cell {
    char *name;
    struct cell *next;
};
```

The straightforward translation into ctypes code would be this, but it does not work:

```python
>>> class cell(Structure):
    ...     _fields_ = [("name", c_char_p),
    ...             ("next", POINTER(cell))]
    ...
Traceback (most recent call last):
  File "<stdin>"", line 1, in ?
  File "<stdin>"", line 2, in cell
NameError: name 'cell' is not defined
```n
because the new `class cell` is not available in the class statement itself. In `ctypes`, we can define the `cell` class and set the `_fields_` attribute later, after the class statement:

```python
>>> from ctypes import *
>>> class cell(Structure):
    ...     pass
    ...
>>> cell._fields_ = [("name", c_char_p),
    ...             ("next", POINTER(cell))]
>>> 
```

Lets try it. We create two instances of `cell`, and let them point to each other, and finally follow the pointer chain a few times:

```python
>>> c1 = cell()
>>> c1.name = "foo"
>>> c2 = cell()
>>> c2.name = "bar"
```
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...    print p.name,
...    p = p.next[0]
...
foo bar foo bar foo bar foo bar

15.17.1.17. Callback functions

c types allows creating C callable function pointers from Python callables. These are sometimes called callback functions.

First, you must create a class for the callback function, the class knows the calling convention, the return type, and the number and types of arguments this function will receive.

The CFUNCTYPE factory function creates types for callback functions using the normal cdecl calling convention, and, on Windows, the WINFUNCTYPE factory function creates types for callback functions using the stdcall calling convention.

Both of these factory functions are called with the result type as first argument, and the callback functions expected argument types as the remaining arguments.

I will present an example here which uses the standard C library's qsort() function, this is used to sort items with the help of a callback function. qsort() will be used to sort an array of integers:

```python
cIntArray5 = c_int * 5
da = IntArray5(5, 1, 7, 33, 99)
qsort = libc.qsort
qsort.restype = None
```
`qsort()` must be called with a pointer to the data to sort, the number of items in the data array, the size of one item, and a pointer to the comparison function, the callback. The callback will then be called with two pointers to items, and it must return a negative integer if the first item is smaller than the second, a zero if they are equal, and a positive integer else.

So our callback function receives pointers to integers, and must return an integer. First we create the `type` for the callback function:

```python
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
```

For the first implementation of the callback function, we simply print the arguments we get, and return 0 (incremental development ;-):

```python
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a, b
...     return 0
... 
>>> 
```

Create the C callable callback:

```python
>>> cmp_func = CMPFUNC(py_cmp_func)
```

And we're ready to go:

```python
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...> 
```
We know how to access the contents of a pointer, so let's redefine our callback:

```python
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a[0], b[0]
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)

Here is what we get on Windows:

```c
qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 7 1
py_cmp_func 33 1
py_cmp_func 99 1
py_cmp_func 5 1
py_cmp_func 7 5
py_cmp_func 33 5
py_cmp_func 99 5
py_cmp_func 7 99
py_cmp_func 33 99
py_cmp_func 7 33
```c

It is funny to see that on Linux the sort function seems to work much more efficiently, it is doing less comparisons:

```c
qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
```c

Ah, we're nearly done! The last step is to actually compare the two items and return a useful result:
```python
def py_cmp_func(a, b):
    ...    print "py_cmp_func", a[0], b[0]
    ...    return a[0] - b[0]
```

Final run on Windows:

```python
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 33 7
py_cmp_func 99 33
py_cmp_func 5 99
py_cmp_func 1 99
py_cmp_func 33 7
py_cmp_func 1 33
py_cmp_func 5 33
py_cmp_func 5 7
py_cmp_func 1 7
py_cmp_func 5 1
```
long as they are used from C code. ctypes doesn’t, and if you don’t, they may be garbage collected, crashing your program when a callback is made.

Also, note that if the callback function is called in a thread created outside of Python’s control (e.g. by the foreign code that calls the callback), ctypes creates a new dummy Python thread on every invocation. This behavior is correct for most purposes, but it means that values stored with threading.local will not survive across different callbacks, even when those calls are made from the same C thread.

15.17.1.18. Accessing values exported from dlls

Some shared libraries not only export functions, they also export variables. An example in the Python library itself is the Py_OptimizeFlag, an integer set to 0, 1, or 2, depending on the -O or -OO flag given on startup.

ctypes can access values like this with the in_dll() class methods of the type. pythonapi is a predefined symbol giving access to the Python C api:

```python
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print opt_flag
  c_long(0)

```

If the interpreter would have been started with -o, the sample would have printed c_long(1), or c_long(2) if -OO would have been specified.

An extended example which also demonstrates the use of pointers accesses the PyImport_FrozenModules pointer exported by Python.
Quoting the Python docs: *This pointer is initialized to point to an array of “struct _frozen” records, terminated by one whose members are all NULL or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.*

So manipulating this pointer could even prove useful. To restrict the example size, we show only how this table can be read with `ctypes`:

```python
>>> from ctypes import *

>>> class struct_frozen(Structure):
    ...    _fields_ = ["name", c_char_p],
    ...            ("code", POINTER(c_ubyte)),
    ...            ("size", c_int)

>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_FrozenModules")

Since `table` is a pointer to the array of `struct_frozen` records, we can iterate over it, but we just have to make sure that our loop terminates, because pointers have no size. Sooner or later it would probably crash with an access violation or whatever, so it's better to break out of the loop when we hit the NULL entry:

```python
>>> for item in table:
    ...    print item.name, item.size
    ...    if item.name is None:
    ...        break
...
__hello__ 104
__phello__ -104
__phello__.spam 104
```
The fact that standard Python has a frozen module and a frozen package (indicated by the negative size member) is not well known, it is only used for testing. Try it out with `import __hello__` for example.

15.17.1.19. Surprises

There are some edge cases in `ctypes` where you might expect something other than what actually happens.

Consider the following example:

```python
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print rc.a.x, rc.a.y, rc.b.x, rc.b.y
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print rc.a.x, rc.a.y, rc.b.x, rc.b.y
3 4 3 4
>>> print
```

Hm. We certainly expected the last statement to print `3 4 1 2`. What happened? Here are the steps of the `rc.a, rc.b = rc.b, rc.a` line above:

```python
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
```
Note that temp0 and temp1 are objects still using the internal buffer of the rc object above. So executing rc.a = temp0 copies the buffer contents of temp0 into rc 's buffer. This, in turn, changes the contents of temp1. So, the last assignment rc.b = temp1, doesn't have the expected effect.

Keep in mind that retrieving sub-objects from Structure, Unions, and Arrays doesn't *copy* the sub-object, instead it retrieves a wrapper object accessing the root-object's underlying buffer.

Another example that may behave different from what one would expect is this:

```python
>>> s = c_char_p()
>>> s.value = "abc def ghi"
>>> s.value
'abc def ghi'
>>> s.value is s.value
False
```

Why is it printing False? ctypes instances are objects containing a memory block plus some descriptors accessing the contents of the memory. Storing a Python object in the memory block does not store the object itself, instead the contents of the object is stored. Accessing the contents again constructs a new Python object each time!

15.17.1.20. Variable-sized data types

ctypes provides some support for variable-sized arrays and structures.

The resize() function can be used to resize the memory buffer of an
existing ctypes object. The function takes the object as first argument, and the requested size in bytes as the second argument. The memory block cannot be made smaller than the natural memory block specified by the objects type, a ValueError is raised if this is tried:

```python
>>> short_array = (c_short * 4)()
>>> print sizeof(short_array)
8
>>> resize(short_array, 4)
Traceback (most recent call last):
  ... ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8

This is nice and fine, but how would one access the additional elements contained in this array? Since the type still only knows about 4 elements, we get errors accessing other elements:

```python
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
  ... IndexError: invalid index
```

Another way to use variable-sized data types with ctypes is to use the dynamic nature of Python, and (re-)define the data type after the required size is already known, on a case by case basis.
When programming in a compiled language, shared libraries are accessed when compiling/linking a program, and when the program is run.

The purpose of the `find_library()` function is to locate a library in a way similar to what the compiler does (on platforms with several versions of a shared library the most recent should be loaded), while the ctypes library loaders act like when a program is run, and call the runtime loader directly.

The `ctypes.util` module provides a function which can help to determine the library to load.

```python
ctypes.util.find_library(name)
```

Try to find a library and return a pathname. `name` is the library name without any prefix like `lib`, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option `-l`). If no library can be found, returns `None`.

The exact functionality is system dependent.

On Linux, `find_library()` tries to run external programs (`/sbin/ldconfig`, `gcc`, and `objdump`) to find the library file. It returns the filename of the library file. Here are some examples:

```python
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
```
On OS X, `find_library()` tries several predefined naming schemes and paths to locate the library, and returns a full pathname if successful:

```python
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
```

On Windows, `find_library()` searches along the system search path, and returns the full pathname, but since there is no predefined naming scheme a call like `find_library("c")` will fail and return `None`.

If wrapping a shared library with `ctypes`, it may be better to determine the shared library name at development time, and hardcode that into the wrapper module instead of using `find_library()` to locate the library at runtime.

## 15.17.2.2. Loading shared libraries

There are several ways to load shared libraries into the Python process. One way is to instantiate one of the following classes:

```python
class ctypes.CDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False, use_last_error=False)
```

Instances of this class represent loaded shared libraries.
Functions in these libraries use the standard C calling convention, and are assumed to return `int`.

```python
class ctypes.OleDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False, use_last_error=False)

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return the windows specific `HRESULT` code. `HRESULT` values contain information specifying whether the function call failed or succeeded, together with additional error code. If the return value signals a failure, an `WindowsError` is automatically raised.
```

```python
class ctypes.WinDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False, use_last_error=False)

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return `int` by default.
```

On Windows CE only the standard calling convention is used, for convenience the `WinDLL` and `OleDLL` use the standard calling convention on this platform.

The Python global interpreter lock is released before calling any function exported by these libraries, and reacquired afterwards.

```python
class ctypes.PyDLL(name, mode=DEFAULT_MODE, handle=None)

Instances of this class behave like `CDLL` instances, except that the Python GIL is not released during the function call, and after the function execution the Python error flag is checked. If the error flag is set, a Python exception is raised.
```

Thus, this is only useful to call Python C api functions directly.

All these classes can be instantiated by calling them with at least
one argument, the pathname of the shared library. If you have an existing handle to an already loaded shared library, it can be passed as the \texttt{handle} named parameter, otherwise the underlying platforms \texttt{dlopen} or \texttt{LoadLibrary} function is used to load the library into the process, and to get a handle to it.

The \textit{mode} parameter can be used to specify how the library is loaded. For details, consult the \texttt{dlopen(3)} manpage. On Windows, \textit{mode} is ignored. On posix systems, RTLD_NOW is always added, and is not configurable.

The \texttt{use_errno} parameter, when set to true, enables a ctypes mechanism that allows accessing the system \texttt{errno} error number in a safe way. ctypes maintains a thread-local copy of the systems \texttt{errno} variable; if you call foreign functions created with \texttt{use_errno=True} then the \texttt{errno} value before the function call is swapped with the ctypes private copy, the same happens immediately after the function call.

The function \texttt{ctypes.get_errno()} returns the value of the ctypes private copy, and the function \texttt{ctypes.set_errno()} changes the ctypes private copy to a new value and returns the former value.

The \texttt{use_last_error} parameter, when set to true, enables the same mechanism for the Windows error code which is managed by the \texttt{GetLastError()} and \texttt{SetLastError()} Windows API functions; \texttt{ctypes.get_last_error()} and \texttt{ctypes.set_last_error()} are used to request and change the ctypes private copy of the windows error code.

\textit{New in version 2.6:} The \texttt{use_last_error} and \texttt{use_errno} optional parameters were added.

c\texttt{types.\texttt{RTLD\_GLOBAL}}
Flag to use as *mode* parameter. On platforms where this flag is not available, it is defined as the integer zero.

**ctypes.**RTLD_LOCAL

Flag to use as *mode* parameter. On platforms where this is not available, it is the same as *RTLD_GLOBAL*.

**ctypes.**DEFAULT_MODE

The default mode which is used to load shared libraries. On OSX 10.3, this is *RTLD_GLOBAL*, otherwise it is the same as *RTLD_LOCAL*.

Instances of these classes have no public methods. Functions exported by the shared library can be accessed as attributes or by index. Please note that accessing the function through an attribute caches the result and therefore accessing it repeatedly returns the same object each time. On the other hand, accessing it through an index returns a new object each time:

```python
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

The following public attributes are available, their name starts with an underscore to not clash with exported function names:

**PyDLL.**_handle

The system handle used to access the library.

**PyDLL.**_name

The name of the library passed in the constructor.

Shared libraries can also be loaded by using one of the prefabricated objects, which are instances of the **LibraryLoader** class, either by calling the **LoadLibrary()** method, or by retrieving the library as attribute of the loader instance.
**class** ctypes.LibraryLoader(dlltype)

Class which loads shared libraries. `dlltype` should be one of the `CDLL`, `PyDLL`, `WinDLL`, or `OleDLL` types.

`__getattr__()` has special behavior: It allows loading a shared library by accessing it as attribute of a library loader instance. The result is cached, so repeated attribute accesses return the same library each time.

**LoadLibrary(name)**

Load a shared library into the process and return it. This method always returns a new instance of the library.

These prefabricated library loaders are available:

**ctypes.cdll**

Creates `CDLL` instances.

**ctypes.windll**

Windows only: Creates `WinDLL` instances.

**ctypes.oledll**

Windows only: Creates `OleDLL` instances.

**ctypes.pydll**

Creates `PyDLL` instances.

For accessing the C Python api directly, a ready-to-use Python shared library object is available:

**ctypes.pythonapi**

An instance of `PyDLL` that exposes Python C API functions as attributes. Note that all these functions are assumed to return C `int`, which is of course not always the truth, so you have to assign the correct `restype` attribute to use these functions.
15.17.2.3. Foreign functions

As explained in the previous section, foreign functions can be accessed as attributes of loaded shared libraries. The function objects created in this way by default accept any number of arguments, accept any ctypes data instances as arguments, and return the default result type specified by the library loader. They are instances of a private class:

```python
class ctypes._FuncPtr
    Base class for C callable foreign functions.
```

Instances of foreign functions are also C compatible data types; they represent C function pointers.

This behavior can be customized by assigning to special attributes of the foreign function object.

**restype**

Assign a ctypes type to specify the result type of the foreign function. Use `None` for `void`, a function not returning anything.

It is possible to assign a callable Python object that is not a ctypes type, in this case the function is assumed to return a C `int`, and the callable will be called with this integer, allowing further processing or error checking. Using this is deprecated, for more flexible post processing or error checking use a ctypes data type as `restype` and assign a callable to the `errcheck` attribute.

**argtypes**

Assign a tuple of ctypes types to specify the argument types that the function accepts. Functions using the `stdcall` calling convention can only be called with the same number of arguments as the length of this tuple; functions using the C
calling convention accept additional, unspecified arguments as well.

When a foreign function is called, each actual argument is passed to the from_param() class method of the items in the argtypes tuple, this method allows adapting the actual argument to an object that the foreign function accepts. For example, a c_char_p item in the argtypes tuple will convert a unicode string passed as argument into a byte string using ctypes conversion rules.

New: It is now possible to put items in argtypes which are not ctypes types, but each item must have a from_param() method which returns a value usable as argument (integer, string, ctypes instance). This allows defining adapters that can adapt custom objects as function parameters.

errcheck
Assign a Python function or another callable to this attribute. The callable will be called with three or more arguments:

callable(result, func, arguments)

result is what the foreign function returns, as specified by the restype attribute.

func is the foreign function object itself, this allows reusing the same callable object to check or post process the results of several functions.

arguments is a tuple containing the parameters originally passed to the function call, this allows specializing the behavior on the arguments used.

The object that this function returns will be returned from the foreign function call, but it can also check the result value and
raise an exception if the foreign function call failed.

```
exception ctypes ArgumentError

This exception is raised when a foreign function call cannot convert one of the passed arguments.
```

### 15.17.2.4. Function prototypes

Foreign functions can also be created by instantiating function prototypes. Function prototypes are similar to function prototypes in C; they describe a function (return type, argument types, calling convention) without defining an implementation. The factory functions must be called with the desired result type and the argument types of the function.

```
cTypes.CFUNCTYPE(restype, *argtypes, use_errno=False, use_last_error=False)
```

The returned function prototype creates functions that use the standard C calling convention. The function will release the GIL during the call. If `use_errno` is set to true, the ctypes private copy of the system `errno` variable is exchanged with the real `errno` value before and after the call; `use_last_error` does the same for the Windows error code.

*Changed in version 2.6:* The optional `use_errno` and `use_last_error` parameters were added.

```
cTypes.WINFUNCTYPE(restype, *argtypes, use_errno=False, use_last_error=False)
```

Windows only: The returned function prototype creates functions that use the `stdcall` calling convention, except on Windows CE where `WINFUNCTYPE()` is the same as `CFUNCTYPE()`. The function will release the GIL during the call. `use_errno` and `use_last_error` have the same meaning as above.
ctypes.\texttt{PYFUNCTYPE}(\texttt{restype}, \*\texttt{argtypes})

The returned function prototype creates functions that use the Python calling convention. The function will \textit{not} release the GIL during the call.

Function prototypes created by these factory functions can be instantiated in different ways, depending on the type and number of the parameters in the call:

\textbf{prototype}(\texttt{address})

Returns a foreign function at the specified address which must be an integer.

\textbf{prototype}(\texttt{callable})

Create a C callable function (a callback function) from a Python \texttt{callable}.

\textbf{prototype}(\texttt{func\_spec[,\ paramflags]})

Returns a foreign function exported by a shared library. \texttt{func\_spec} must be a 2-tuple (\texttt{name\_or\_ordinal,\ library}). The first item is the name of the exported function as string, or the ordinal of the exported function as small integer. The second item is the shared library instance.

\textbf{prototype}(\texttt{vtbl\_index,\ name[,\ paramflags[,\ iid]]})

Returns a foreign function that will call a COM method. \texttt{vtbl\_index} is the index into the virtual function table, a small non-negative integer. \texttt{name} is name of the COM method. \texttt{iid} is an optional pointer to the interface identifier which is used in extended error reporting.

COM methods use a special calling convention: They require a pointer to the COM interface as first argument, in addition to those parameters that are specified in the \texttt{argtypes} tuple.
The optional *paramflags* parameter creates foreign function wrappers with much more functionality than the features described above.

*paramflags* must be a tuple of the same length as *argtypes*.

Each item in this tuple contains further information about a parameter, it must be a tuple containing one, two, or three items.

The first item is an integer containing a combination of direction flags for the parameter:

1
   Specifies an input parameter to the function.
2
   Output parameter. The foreign function fills in a value.
4
   Input parameter which defaults to the integer zero.

The optional second item is the parameter name as string. If this is specified, the foreign function can be called with named parameters.

The optional third item is the default value for this parameter.

This example demonstrates how to wrap the Windows *MessageBoxA* function so that it supports default parameters and named arguments. The C declaration from the windows header file is this:

```c
WINUSERAPI int WINAPI MessageBoxA(
    HWND hWnd,
    LPCSTR lpText,
    LPCSTR lpCaption,
    UINT uType);
```
Here is the wrapping with `ctypes`:

```python
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCSTR, LPCSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption")
>>> MessageBox = prototype(("MessageBoxA", windll.user32), paramflags)
```  

The `MessageBox` foreign function can now be called in these ways:

```python
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
>>> 
```

A second example demonstrates output parameters. The `win32 GetWindowRect` function retrieves the dimensions of a specified window by copying them into `RECT` structure that the caller has to supply. Here is the C declaration:

```text
WINUSERAPI BOOL WINAPI GetWindowRect( 
    HWND hWnd, 
    LPRECT lpRect);
```

Here is the wrapping with `ctypes`:

```python
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)

```  

Functions with output parameters will automatically return the output parameter value if there is a single one, or a tuple containing the output parameter values when there are more than one, so the
GetWindowRect function now returns a RECT instance, when called.

Output parameters can be combined with the `errcheck` protocol to do further output processing and error checking. The win32 `GetWindowRect` api function returns a `BOOL` to signal success or failure, so this function could do the error checking, and raises an exception when the api call failed:

```python
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
... >>> GetWindowRect.errcheck = errcheck
```  
If the `errcheck` function returns the argument tuple it receives unchanged, `ctypes` continues the normal processing it does on the output parameters. If you want to return a tuple of window coordinates instead of a RECT instance, you can retrieve the fields in the function and return them instead, the normal processing will no longer take place:

```python
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
... >>> GetWindowRect.errcheck = errcheck
```  

15.17.2.5. Utility functions

`ctypes.addressof(obj)`

Returns the address of the memory buffer as integer. `obj` must be an instance of a ctypes type.
**ctypes.** `alignment(obj_or_type)`

Returns the alignment requirements of a ctypes type. `obj_or_type` must be a ctypes type or instance.

**ctypes.** `byref(obj[, offset])`

Returns a light-weight pointer to `obj`, which must be an instance of a ctypes type. `offset` defaults to zero, and must be an integer that will be added to the internal pointer value.

`byref(obj, offset)` corresponds to this C code:

```c
(((char *)&obj) + offset)
```

The returned object can only be used as a foreign function call parameter. It behaves similar to `pointer(obj)`, but the construction is a lot faster.

*New in version 2.6*: The `offset` optional argument was added.

**ctypes.** `cast(obj, type)`

This function is similar to the cast operator in C. It returns a new instance of `type` which points to the same memory block as `obj`. `type` must be a pointer type, and `obj` must be an object that can be interpreted as a pointer.

**ctypes.** `create_string_buffer(init_or_size[, size])`

This function creates a mutable character buffer. The returned object is a ctypes array of `c_char`.

`init_or_size` must be an integer which specifies the size of the array, or a string which will be used to initialize the array items.

If a string is specified as first argument, the buffer is made one item larger than the length of the string so that the last element in the array is a NUL termination character. An integer can be
passed as second argument which allows specifying the size of the array if the length of the string should not be used.

If the first parameter is a unicode string, it is converted into an 8-bit string according to ctypes conversion rules.

cryptypes.\texttt{create\_unicode\_buffer}(\texttt{init\_or\_size[, size]})

This function creates a mutable unicode character buffer. The returned object is a ctypes array of \texttt{c\_wchar}.

\texttt{init\_or\_size} must be an integer which specifies the size of the array, or a unicode string which will be used to initialize the array items.

If a unicode string is specified as first argument, the buffer is made one item larger than the length of the string so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the string should not be used.

If the first parameter is an 8-bit string, it is converted into a unicode string according to ctypes conversion rules.

cryptypes.\texttt{DllCanUnloadNow}()

Windows only: This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllCanUnloadNow function that the \texttt{_ctypes} extension dll exports.

cryptypes.\texttt{DllGetClassObject}()

Windows only: This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllGetClassObject function that the \texttt{_ctypes} extension dll exports.

cryptypes.util.\texttt{find\_library}(\texttt{name})

Try to find a library and return a pathname. \texttt{name} is the library
name without any prefix like `lib`, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option `-l`). If no library can be found, returns `None`.

The exact functionality is system dependent.

*Changed in version 2.6:* Windows only: `find_library("m")` or `find_library("c")` return the result of a call to `find_msvcrt()`.

```python
cTYPES.util.find_msvcrt()
```

Windows only: return the filename of the VC runtime library used by Python, and by the extension modules. If the name of the library cannot be determined, `None` is returned.

If you need to free memory, for example, allocated by an extension module with a call to the `free(void *)`, it is important that you use the function in the same library that allocated the memory.

*New in version 2.6.*

```python
cTYPES.FormatError([code])
```

Windows only: Returns a textual description of the error code `code`. If no error code is specified, the last error code is used by calling the Windows api function GetLastError.

```python
cTYPES.GetLastError()
```

Windows only: Returns the last error code set by Windows in the calling thread. This function calls the Windows `GetLastError()` function directly, it does not return the ctypes-private copy of the error code.

```python
cTYPES.get_errno()
```

Returns the current value of the ctypes-private copy of the system `errno` variable in the calling thread.
New in version 2.6.

`ctypes.get_last_error()`
Windows only: returns the current value of the ctypes-private copy of the system `LastError` variable in the calling thread.

New in version 2.6.

`ctypes.memmove(dst, src, count)`
Same as the standard C memmove library function: copies `count` bytes from `src` to `dst`. `dst` and `src` must be integers or ctypes instances that can be converted to pointers.

`ctypes.memset(dst, c, count)`
Same as the standard C memset library function: fills the memory block at address `dst` with `count` bytes of value `c`. `dst` must be an integer specifying an address, or a ctypes instance.

`ctypes.POINTER(type)`
This factory function creates and returns a new ctypes pointer type. Pointer types are cached and reused internally, so calling this function repeatedly is cheap. `type` must be a ctypes type.

`ctypes.pointer(obj)`
This function creates a new pointer instance, pointing to `obj`. The returned object is of the type `POINTER(type(obj))`.

Note: If you just want to pass a pointer to an object to a foreign function call, you should use `byref(obj)` which is much faster.

`ctypes.resize(obj, size)`
This function resizes the internal memory buffer of `obj`, which must be an instance of a ctypes type. It is not possible to make the buffer smaller than the native size of the objects type, as given by `sizeof(type(obj))`, but it is possible to enlarge the
buffer.

**ctypes.set_conversion_mode(encoding, errors)**

This function sets the rules that ctypes objects use when converting between 8-bit strings and unicode strings. `encoding` must be a string specifying an encoding, like 'utf-8' or 'mbcs', `errors` must be a string specifying the error handling on encoding/decoding errors. Examples of possible values are "strict", "replace", or "ignore".

`set_conversion_mode()` returns a 2-tuple containing the previous conversion rules. On windows, the initial conversion rules are ('mbcs', 'ignore'), on other systems ('ascii', 'strict').

**ctypes.set_errno(value)**

Set the current value of the ctypes-private copy of the system `errno` variable in the calling thread to `value` and return the previous value.

*New in version 2.6.*

**ctypes.set_last_error(value)**

Windows only: set the current value of the ctypes-private copy of the system `LastError` variable in the calling thread to `value` and return the previous value.

*New in version 2.6.*

**ctypes.sizeof(obj_or_type)**

Returns the size in bytes of a ctypes type or instance memory buffer. Does the same as the C `sizeof` operator.

**ctypes.string_at(address[, size])**

This function returns the string starting at memory address `address`. If size is specified, it is used as size, otherwise the
string is assumed to be zero-terminated.

cypes.WinError(code=None, descr=None)
Windows only: this function is probably the worst-named thing in
cypes. It creates an instance of WindowsError. If code is not
specified, GetLastError is called to determine the error code. If
descr is not specified, FormatError() is called to get a textual
description of the error.

cypes.wstring_at(address[, size])
This function returns the wide character string starting at memory
address address as unicode string. If size is specified, it is used
as the number of characters of the string, otherwise the string is
assumed to be zero-terminated.

15.17.2.6. Data types

class cypes._CData
This non-public class is the common base class of all cypes data
types. Among other things, all cypes type instances contain a
memory block that hold C compatible data; the address of the
memory block is returned by the addressof() helper function.
Another instance variable is exposed as _objects; this contains
other Python objects that need to be kept alive in case the
memory block contains pointers.

Common methods of cypes data types, these are all class
methods (to be exact, they are methods of the metaclass):

from_buffer(source[, offset])
This method returns a cypes instance that shares the buffer
of the source object. The source object must support the
writeable buffer interface. The optional offset parameter
specifies an offset into the source buffer in bytes; the default
is zero. If the source buffer is not large enough a `ValueError` is raised.

*New in version 2.6.*

`from_buffer_copy(source[, offset])`
This method creates a ctypes instance, copying the buffer from the `source` object buffer which must be readable. The optional `offset` parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a `ValueError` is raised.

*New in version 2.6.*

`from_address(address)`
This method returns a ctypes type instance using the memory specified by `address` which must be an integer.

`from_param(obj)`
This method adapts `obj` to a ctypes type. It is called with the actual object used in a foreign function call when the type is present in the foreign function’s `argtypes` tuple; it must return an object that can be used as a function call parameter.

All ctypes data types have a default implementation of this classmethod that normally returns `obj` if that is an instance of the type. Some types accept other objects as well.

`in_dll(library, name)`
This method returns a ctypes type instance exported by a shared library. `name` is the name of the symbol that exports the data, `library` is the loaded shared library.

Common instance variables of ctypes data types:
**_b_base_**
Sometimes ctypes data instances do not own the memory block they contain, instead they share part of the memory block of a base object. The _b_base_ read-only member is the root ctypes object that owns the memory block.

**_b_needsfree_**
This read-only variable is true when the ctypes data instance has allocated the memory block itself, false otherwise.

**_objects**
This member is either None or a dictionary containing Python objects that need to be kept alive so that the memory block contents is kept valid. This object is only exposed for debugging; never modify the contents of this dictionary.

### 15.17.2.7. Fundamental data types

**class ctypes._SimpleCData**
This non-public class is the base class of all fundamental ctypes data types. It is mentioned here because it contains the common attributes of the fundamental ctypes data types. _SimpleCData is a subclass of _CData, so it inherits their methods and attributes.

*Changed in version 2.6:* ctypes data types that are not and do not contain pointers can now be pickled.

Instances have a single attribute:

**value**
This attribute contains the actual value of the instance. For integer and pointer types, it is an integer, for character types, it is a single character string, for character pointer types it is a Python string or unicode string.
When the \texttt{value} attribute is retrieved from a ctypes instance, usually a new object is returned each time. \texttt{ctypes} does \textit{not} implement original object return, always a new object is constructed. The same is true for all other ctypes object instances.

Fundamental data types, when returned as foreign function call results, or, for example, by retrieving structure field members or array items, are transparently converted to native Python types. In other words, if a foreign function has a restype of \texttt{c\_char\_p}, you will always receive a Python string, \textit{not} a \texttt{c\_char\_p} instance.

Subclasses of fundamental data types do \textit{not} inherit this behavior. So, if a foreign functions restype is a subclass of \texttt{c\_void\_p}, you will receive an instance of this subclass from the function call. Of course, you can get the value of the pointer by accessing the \texttt{value} attribute.

These are the fundamental ctypes data types:

\begin{description}
\item[\texttt{class ctypes.c\_byte}] Represents the C \texttt{signed char} datatype, and interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.
\item[\texttt{class ctypes.c\_char}] Represents the C \texttt{char} datatype, and interprets the value as a single character. The constructor accepts an optional string initializer, the length of the string must be exactly one character.
\item[\texttt{class ctypes.c\_char\_p}] Represents the C \texttt{char \*} datatype when it points to a zero-terminated string. For a general character pointer that may also point to binary data, \texttt{POINTER(c\_char)} must be used. The constructor accepts an integer address, or a string.
\end{description}
class ctypes.c_double
Represents the C double datatype. The constructor accepts an optional float initializer.

class ctypes.c_longdouble
Represents the C long double datatype. The constructor accepts an optional float initializer. On platforms where \texttt{sizeof(long double)} == \texttt{sizeof(double)} it is an alias to \texttt{c_double}.

\textit{New in version 2.6.}

class ctypes.c_float
Represents the C float datatype. The constructor accepts an optional float initializer.

class ctypes.c_int
Represents the C signed int datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where \texttt{sizeof(int)} == \texttt{sizeof(long)} it is an alias to \texttt{c_long}.

class ctypes.c_int8
Represents the C 8-bit signed int datatype. Usually an alias for \texttt{c_byte}.

class ctypes.c_int16
Represents the C 16-bit signed int datatype. Usually an alias for \texttt{c_short}.

class ctypes.c_int32
Represents the C 32-bit signed int datatype. Usually an alias for \texttt{c_int}.

class ctypes.c_int64
Represents the C 64-bit *signed* *int* datatype. Usually an alias for `c_longlong`.

*class ctypes.c_long*

Represents the C *signed* *long* datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

*class ctypes.c_longlong*

Represents the C *signed* *long* *long* datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

*class ctypes.c_short*

Represents the C *signed* *short* datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

*class ctypes.c_size_t*

Represents the C *size_t* datatype.

*class ctypes.c_ssize_t*

Represents the C *ssize_t* datatype.

*New in version 2.7.*

*class ctypes.c_ubyte*

Represents the C *unsigned* *char* datatype, it interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

*class ctypes.c_uint*

Represents the C *unsigned* *int* datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an
alias for `c_ulong`.

```python
class ctypes.c_uint8
    Represents the C 8-bit unsigned `int` datatype. Usually an alias for `c_ubyte`.
```

```python
class ctypes.c_uint16
    Represents the C 16-bit unsigned `int` datatype. Usually an alias for `c_ushort`.
```

```python
class ctypes.c_uint32
    Represents the C 32-bit unsigned `int` datatype. Usually an alias for `c_uint`.
```

```python
class ctypes.c_uint64
    Represents the C 64-bit unsigned `int` datatype. Usually an alias for `c_ulonglong`.
```

```python
class ctypes.c_ulong
    Represents the C unsigned `long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.
```

```python
class ctypes.c_ulonglong
    Represents the C unsigned `long long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.
```

```python
class ctypes.c_ushort
    Represents the C unsigned `short` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.
```

```python
class ctypes.c_void_p
Represents the C `void *` type. The value is represented as integer. The constructor accepts an optional integer initializer.

```python
class ctypes.c_wchar
```
Represents the C `wchar_t` datatype, and interprets the value as a single character unicode string. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

```python
class ctypes.c_wchar_p
```
Represents the C `wchar_t *` datatype, which must be a pointer to a zero-terminated wide character string. The constructor accepts an integer address, or a string.

```python
class ctypes.c_bool
```
Represent the C `bool` datatype (more accurately, `_Bool` from C99). Its value can be `True` or `False`, and the constructor accepts any object that has a truth value.

*New in version 2.6.*

```python
class ctypes.HRESULT
```
Windows only: Represents a `HRESULT` value, which contains success or error information for a function or method call.

```python
class ctypes.py_object
```
Represents the C `PyObject *` datatype. Calling this without an argument creates a `NULL` `PyObject *` pointer.

The `ctypes.wintypes` module provides quite some other Windows specific data types, for example `HWND`, `WPARAM`, or `DWORD`. Some useful structures like `MSG` or `RECT` are also defined.

15.17.2.8. Structured data types
class ctypes.Union(*args, **kw)
    Abstract base class for unions in native byte order.

class ctypes.BigEndianStructure(*args, **kw)
    Abstract base class for structures in big endian byte order.

class ctypes.LittleEndianStructure(*args, **kw)
    Abstract base class for structures in little endian byte order.

Structures with non-native byte order cannot contain pointer type fields, or any other data types containing pointer type fields.

class ctypes.Structure(*args, **kw)
    Abstract base class for structures in native byte order.

Concrete structure and union types must be created by subclassing one of these types, and at least define a _fields_ class variable. ctypes will create descriptors which allow reading and writing the fields by direct attribute accesses. These are the _fields_

A sequence defining the structure fields. The items must be 2-tuples or 3-tuples. The first item is the name of the field, the second item specifies the type of the field; it can be any ctypes data type.

For integer type fields like c_int, a third optional item can be given. It must be a small positive integer defining the bit width of the field.

Field names must be unique within one structure or union. This is not checked, only one field can be accessed when names are repeated.

It is possible to define the _fields_ class variable after the class statement that defines the Structure subclass, this
allows creating data types that directly or indirectly reference themselves:

```python
class List(Structure):
    pass
List._fields_ = ["pnext", POINTER(List)], ...
```

The `_fields_` class variable must, however, be defined before the type is first used (an instance is created, `sizeof()` is called on it, and so on). Later assignments to the `_fields_` class variable will raise an AttributeError.

It is possible to defined sub-subclasses of structure types, they inherit the fields of the base class plus the `_fields_` defined in the sub-subclass, if any.

`_pack_`
An optional small integer that allows overriding the alignment of structure fields in the instance. `_pack_` must already be defined when `_fields_` is assigned, otherwise it will have no effect.

`_anonymous_`
An optional sequence that lists the names of unnamed (anonymous) fields. `_anonymous_` must be already defined when `_fields_` is assigned, otherwise it will have no effect.

The fields listed in this variable must be structure or union type fields. `ctypes` will create descriptors in the structure type that allow accessing the nested fields directly, without the need to create the structure or union field.

Here is an example type (Windows):
The `TYPEDESC` structure describes a COM data type, the `vt` field specifies which one of the union fields is valid. Since the `u` field is defined as anonymous field, it is now possible to access the members directly off the `TYPEDESC` instance. `td.lptdesc` and `td.u.lptdesc` are equivalent, but the former is faster since it does not need to create a temporary union instance:

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

It is possible to defined sub-subclasses of structures, they inherit the fields of the base class. If the subclass definition has a separate `_fields_` variable, the fields specified in this are appended to the fields of the base class.

Structure and union constructors accept both positional and keyword arguments. Positional arguments are used to initialize member fields in the same order as they are appear in `_fields_`. Keyword arguments in the constructor are interpreted as attribute assignments, so they will initialize `_fields_` with the same name, or create new attributes for names not present in `_fields_`.

15.17.2.9. Arrays and pointers
class ctypes.Array(*args)

Abstract base class for arrays.

The recommended way to create concrete array types is by multiplying any ctypes data type with a positive integer. Alternatively, you can subclass this type and define _length_ and _type_ class variables. Array elements can be read and written using standard subscript and slice accesses; for slice reads, the resulting object is not itself an Array.

_length_

A positive integer specifying the number of elements in the array. Out-of-range subscripts result in an IndexError. Will be returned by len().

_type_

Specifies the type of each element in the array.

Array subclass constructors accept positional arguments, used to initialize the elements in order.

class ctypes._Pointer

Private, abstract base class for pointers.

Concrete pointer types are created by calling POINTER() with the type that will be pointed to; this is done automatically by pointer().

If a pointer points to an array, its elements can be read and written using standard subscript and slice accesses. Pointer objects have no size, so len() will raise TypeError. Negative subscripts will read from the memory before the pointer (as in C), and out-of-range subscripts will probably crash with an access violation (if you’re lucky).
__type__

Specifies the type pointed to.

contents

Returns the object to which the pointer points. Assigning to this attribute changes the pointer to point to the assigned object.
16. Optional Operating System Services

The modules described in this chapter provide interfaces to operating system features that are available on selected operating systems only. The interfaces are generally modeled after the Unix or C interfaces but they are available on some other systems as well (e.g. Windows or NT). Here’s an overview:

- **16.1. select — Waiting for I/O completion**
  - 16.1.1. Edge and Level Trigger Polling (epoll) Objects
  - 16.1.2. Polling Objects
  - 16.1.3. Kqueue Objects
  - 16.1.4. Kevent Objects

- **16.2. threading — Higher-level threading interface**
  - 16.2.1. Thread Objects
  - 16.2.2. Lock Objects
  - 16.2.3. RLock Objects
  - 16.2.4. Condition Objects
  - 16.2.5. Semaphore Objects
    - 16.2.5.1. Semaphore Example
  - 16.2.6. Event Objects
  - 16.2.7. Timer Objects
  - 16.2.8. Using locks, conditions, and semaphores in the with statement
  - 16.2.9. Importing in threaded code

- **16.3. thread — Multiple threads of control**

- **16.4. dummy_threading — Drop-in replacement for the threading module**

- **16.5. dummy_thread — Drop-in replacement for the thread module**

- **16.6. multiprocessing — Process-based “threading” interface**
16.6.1. Introduction

- 16.6.1.1. The Process class
- 16.6.1.2. Exchanging objects between processes
- 16.6.1.3. Synchronization between processes
- 16.6.1.4. Sharing state between processes
- 16.6.1.5. Using a pool of workers

16.6.2. Reference

- 16.6.2.1. Process and exceptions
- 16.6.2.2. Pipes and Queues
- 16.6.2.3. Miscellaneous
- 16.6.2.4. Connection Objects
- 16.6.2.5. Synchronization primitives
- 16.6.2.6. Shared ctypes Objects
  - 16.6.2.6.1. The multiprocessing.sharedctypes module
- 16.6.2.7. Managers
  - 16.6.2.7.1. Customized managers
  - 16.6.2.7.2. Using a remote manager
- 16.6.2.8. Proxy Objects
  - 16.6.2.8.1. Cleanup
- 16.6.2.9. Process Pools
- 16.6.2.10. Listeners and Clients
  - 16.6.2.10.1. Address Formats
- 16.6.2.11. Authentication keys
- 16.6.2.12. Logging
- 16.6.2.13. The multiprocessing.dummy module

16.6.3. Programming guidelines

- 16.6.3.1. All platforms
- 16.6.3.2. Windows

16.6.4. Examples

- 16.7. mmap — Memory-mapped file support
- 16.8. readline — GNU readline interface
  - 16.8.1. Init file
• 16.8.2. Line buffer
• 16.8.3. History file
• 16.8.4. History list
• 16.8.5. Startup hooks
• 16.8.6. Completion
• 16.8.7. Example
• 16.9. rlcompleter — Completion function for GNU readline
  • 16.9.1. Completer Objects
16.1. select — Waiting for I/O completion

This module provides access to the `select()` and `poll()` functions available in most operating systems, `epoll()` available on Linux 2.5+ and `kqueue()` available on most BSD. Note that on Windows, it only works for sockets; on other operating systems, it also works for other file types (in particular, on Unix, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

The module defines the following:

`exception` `select.error`

The exception raised when an error occurs. The accompanying value is a pair containing the numeric error code from `errno` and the corresponding string, as would be printed by the C function `perror()`.

`select.epoll([sizehint=-1])`

(Only supported on Linux 2.5.44 and newer.) Returns an edge polling object, which can be used as Edge or Level Triggered interface for I/O events; see section Edge and Level Trigger Polling (epoll) Objects below for the methods supported by epolling objects.

*New in version 2.6.*

`select.poll()`

(Not supported by all operating systems.) Returns a polling object, which supports registering and unregistering file descriptors, and then polling them for I/O events; see section Polling Objects below for the methods supported by polling
objects.

select. **kqueue**()
(Only supported on BSD.) Returns a kernel queue object; see section *Kqueue Objects* below for the methods supported by kqueue objects.

*New in version 2.6.*

select. **kevent**(ident, filter=KQ_FILTER_READ, flags=KQ_EV_ADD, fflags=0, data=0, udata=0)
(Only supported on BSD.) Returns a kernel event object; see section *Kevent Objects* below for the methods supported by kevent objects.

*New in version 2.6.*

select. **select**(rlist, wlist, xlist[, timeout])
This is a straightforward interface to the Unix select() system call. The first three arguments are sequences of ‘waitable objects’: either integers representing file descriptors or objects with a parameterless method named *fileno()* returning such an integer:

- *rlist*: wait until ready for reading
- *wlist*: wait until ready for writing
- *xlist*: wait for an “exceptional condition” (see the manual page for what your system considers such a condition)

Empty sequences are allowed, but acceptance of three empty sequences is platform-dependent. (It is known to work on Unix but not on Windows.) The optional *timeout* argument specifies a time-out as a floating point number in seconds. When the *timeout* argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.
The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Among the acceptable object types in the sequences are Python file objects (e.g. `sys.stdin`, or objects returned by `open()` or `os.popen()`), socket objects returned by `socket.socket()`. You may also define a *wrapper* class yourself, as long as it has an appropriate `fileno()` method (that really returns a file descriptor, not just a random integer).

**Note:** File objects on Windows are not acceptable, but sockets are. On Windows, the underlying `select()` function is provided by the WinSock library, and does not handle file descriptors that don't originate from WinSock.

**select.PIPE_BUF**

Files reported as ready for writing by `select()`, `poll()` or similar interfaces in this module are guaranteed to not block on a write of up to `PIPE_BUF` bytes. This value is guaranteed by POSIX to be at least 512. Availability: Unix.

*New in version 2.7.*
16.1.1. Edge and Level Trigger Polling (epoll) Objects

http://linux.die.net/man/4/epoll

eventmask

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPOLLIN</td>
<td>Available for read</td>
</tr>
<tr>
<td>EPOLLOUT</td>
<td>Available for write</td>
</tr>
<tr>
<td>EPOLLPRI</td>
<td>Urgent data for read</td>
</tr>
<tr>
<td>EPOLLERR</td>
<td>Error condition happened on the assoc. fd</td>
</tr>
<tr>
<td>EPOLLHUP</td>
<td>Hang up happened on the assoc. fd</td>
</tr>
<tr>
<td>EPOLLET</td>
<td>Set Edge Trigger behavior, the default is Level Trigger behavior</td>
</tr>
<tr>
<td>EPOLLONESHOT</td>
<td>Set one-shot behavior. After one event is pulled out, the fd is internally disabled</td>
</tr>
<tr>
<td>EPOLLRDNORM</td>
<td>Equivalent to EPOLLIN</td>
</tr>
<tr>
<td>EPOLLRDBAND</td>
<td>Priority data band can be read.</td>
</tr>
<tr>
<td>EPOLLWRNORM</td>
<td>Equivalent to EPOLLOUT</td>
</tr>
<tr>
<td>EPOLLWRBAND</td>
<td>Priority data may be written.</td>
</tr>
<tr>
<td>EPOLLMMSG</td>
<td>Ignored.</td>
</tr>
</tbody>
</table>

epoll.close()      
Close the control file descriptor of the epoll object.

epoll.fileno()     
Return the file descriptor number of the control fd.

epoll.fromfd(fd)   
Create an epoll object from a given file descriptor.
epoll.register(fd[, eventmask])
Register a fd descriptor with the epoll object.

**Note:** Registering a file descriptor that’s already registered raises an IOError – contrary to Polling Objects's register.

epoll.modify(fd, eventmask)
Modify a register file descriptor.

epoll.unregister(fd)
Remove a registered file descriptor from the epoll object.

epoll.poll([timeout=-1[, maxevents=-1]])
Wait for events. timeout in seconds (float)
16.1.2. Polling Objects

The `poll()` system call, supported on most Unix systems, provides better scalability for network servers that service many, many clients at the same time. `poll()` scales better because the system call only requires listing the file descriptors of interest, while `select()` builds a bitmap, turns on bits for the fds of interest, and then afterward the whole bitmap has to be linearly scanned again. `select()` is $O(\text{highest file descriptor})$, while `poll()` is $O(\text{number of file descriptors})$.

`poll.register(fd[, eventmask])`

Register a file descriptor with the polling object. Future calls to the `poll()` method will then check whether the file descriptor has any pending I/O events. `fd` can be either an integer, or an object with a `fileno()` method that returns an integer. File objects implement `fileno()`, so they can also be used as the argument.

`eventmask` is an optional bitmask describing the type of events you want to check for, and can be a combination of the constants `POLLIN`, `POLLPRI`, and `POLLOUT`, described in the table below. If not specified, the default value used will check for all 3 types of events.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLLIN</td>
<td>There is data to read</td>
</tr>
<tr>
<td>POLLPRI</td>
<td>There is urgent data to read</td>
</tr>
<tr>
<td>POLLOUT</td>
<td>Ready for output: writing will not block</td>
</tr>
<tr>
<td>POLLERR</td>
<td>Error condition of some sort</td>
</tr>
<tr>
<td>POLLHUP</td>
<td>Hung up</td>
</tr>
<tr>
<td>POLLNVAL</td>
<td>Invalid request: descriptor not open</td>
</tr>
</tbody>
</table>
Registering a file descriptor that’s already registered is not an error, and has the same effect as registering the descriptor exactly once.

**poll. modify**(fd, eventmask)

Modifies an already registered fd. This has the same effect as `register(fd, eventmask)`. Attempting to modify a file descriptor that was never registered causes an IOError exception with errno ENOENT to be raised.

*New in version 2.6.*

**poll. unregister**(fd)

Remove a file descriptor being tracked by a polling object. Just like the `register()` method, `fd` can be an integer or an object with a `fileno()` method that returns an integer.

Attempting to remove a file descriptor that was never registered causes a KeyError exception to be raised.

**poll. poll**([timeout])

Polls the set of registered file descriptors, and returns a possibly-empty list containing (fd, event) 2-tuples for the descriptors that have events or errors to report. `fd` is the file descriptor, and `event` is a bitmask with bits set for the reported events for that descriptor — POLLIN for waiting input, POLLOUT to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If `timeout` is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If `timeout` is omitted, negative, or None, the call will block until there is an event for this poll object.
16.1.3. Kqueue Objects

kqueue. close()
   Close the control file descriptor of the kqueue object.

kqueue. fileno()
   Return the file descriptor number of the control fd.

kqueue. fromfd(fd)
   Create a kqueue object from a given file descriptor.

kqueue. control(changelist, max_events[, timeout=None]) → eventlist
   Low level interface to kevent

   - changelist must be an iterable of kevent object or None
   - max_events must be 0 or a positive integer
   - timeout in seconds (floats possible)
16.1.4. Kevent Objects

https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

**kevent**: `ident`

Value used to identify the event. The interpretation depends on the filter but it's usually the file descriptor. In the constructor `ident` can either be an int or an object with a `fileno()` function. `kevent` stores the integer internally.

**kevent**: `filter`

Name of the kernel filter.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>KQ_FILTER_READ</code></td>
<td>Takes a descriptor and returns whenever there is data available to read</td>
</tr>
<tr>
<td><code>KQ_FILTER_WRITE</code></td>
<td>Takes a descriptor and returns whenever there is data available to write</td>
</tr>
<tr>
<td><code>KQ_FILTER_AIO</code></td>
<td>AIO requests</td>
</tr>
<tr>
<td><code>KQ_FILTER_VNODE</code></td>
<td>Returns when one or more of the requested events watched in <code>fflag</code> occurs</td>
</tr>
<tr>
<td><code>KQ_FILTER_PROC</code></td>
<td>Watch for events on a process id</td>
</tr>
<tr>
<td><code>KQ_FILTER_NETDEV</code></td>
<td>Watch for events on a network device [not available on Mac OS X]</td>
</tr>
<tr>
<td><code>KQ_FILTER_SIGNAL</code></td>
<td>Returns whenever the watched signal is delivered to the process</td>
</tr>
<tr>
<td><code>KQ_FILTER_TIMER</code></td>
<td>Establishes an arbitrary timer</td>
</tr>
</tbody>
</table>

**kevent**: `flags`

Filter action.
<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KQ_EV_ADD</td>
<td>Adds or modifies an event</td>
</tr>
<tr>
<td>KQ_EV_DELETE</td>
<td>Removes an event from the queue</td>
</tr>
<tr>
<td>KQ_EV_ENABLE</td>
<td>Permits control() to returns the event</td>
</tr>
<tr>
<td>KQ_EV_DISABLE</td>
<td>Disable event</td>
</tr>
<tr>
<td>KQ_EV_ONESHOT</td>
<td>Removes event after first occurrence</td>
</tr>
<tr>
<td>KQ_EV_CLEAR</td>
<td>Reset the state after an event is retrieved</td>
</tr>
<tr>
<td>KQ_EV_SYSFLAGS</td>
<td>Internal event</td>
</tr>
<tr>
<td>KQ_EV_FLAG1</td>
<td>Internal event</td>
</tr>
<tr>
<td>KQ_EV_EOF</td>
<td>Filter specific EOF condition</td>
</tr>
<tr>
<td>KQ_EV_ERROR</td>
<td>See return values</td>
</tr>
</tbody>
</table>

**kevent.fflags**

Filter specific flags.

KQ_FILTER_READ and KQ_FILTER_WRITE filter flags:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KQ_NOTE_LOWAT</td>
<td>low water mark of a socket buffer</td>
</tr>
</tbody>
</table>

KQ_FILTER_VNODE filter flags:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KQ_NOTE_DELETE</td>
<td><code>unlink()</code> was called</td>
</tr>
<tr>
<td>KQ_NOTE_WRITE</td>
<td>a write occurred</td>
</tr>
<tr>
<td>KQ_NOTE_EXTEND</td>
<td>the file was extended</td>
</tr>
<tr>
<td>KQ_NOTE_ATTRIB</td>
<td>an attribute was changed</td>
</tr>
<tr>
<td>KQ_NOTE_LINK</td>
<td>the link count has changed</td>
</tr>
<tr>
<td>KQ_NOTE_RENAME</td>
<td>the file was renamed</td>
</tr>
<tr>
<td>KQ_NOTE_REVOKE</td>
<td>access to the file was revoked</td>
</tr>
</tbody>
</table>

KQ_FILTER_PROC filter flags:
<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KQ_NOTE_EXIT</td>
<td>the process has exited</td>
</tr>
<tr>
<td>KQ_NOTE_FORK</td>
<td>the process has called fork()</td>
</tr>
<tr>
<td>KQ_NOTE_EXEC</td>
<td>the process has executed a new process</td>
</tr>
<tr>
<td>KQ_NOTE_PCTRLMASK</td>
<td>internal filter flag</td>
</tr>
<tr>
<td>KQ_NOTE_PDATAMASK</td>
<td>internal filter flag</td>
</tr>
<tr>
<td>KQ_NOTE_TRACK</td>
<td>follow a process across fork()</td>
</tr>
<tr>
<td>KQ_NOTE_CHILD</td>
<td>returned on the child process for NOTE_TRACK</td>
</tr>
<tr>
<td>KQ_NOTE_TRACKERR</td>
<td>unable to attach to a child</td>
</tr>
</tbody>
</table>

KQ_FILTER_NETDEV filter flags (not available on Mac OS X):

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KQ_NOTE_LINKUP</td>
<td>link is up</td>
</tr>
<tr>
<td>KQ_NOTE_LINKDOWN</td>
<td>link is down</td>
</tr>
<tr>
<td>KQ_NOTE_LINKINV</td>
<td>link state is invalid</td>
</tr>
</tbody>
</table>

kevent.data
Filter specific data.

kevent.udata
User defined value.
16.2. threading — Higher-level threading interface

Source code: Lib/threading.py

This module constructs higher-level threading interfaces on top of the lower level thread module. See also the mutex and Queue modules.

The dummy_threading module is provided for situations where threading cannot be used because thread is missing.

**Note:** Starting with Python 2.6, this module provides PEP 8 compliant aliases and properties to replace the camelCase names that were inspired by Java's threading API. This updated API is compatible with that of the multiprocessing module. However, no schedule has been set for the deprecation of the camelCase names and they remain fully supported in both Python 2.x and 3.x.

**Note:** Starting with Python 2.5, several Thread methods raise RuntimeError instead of AssertionError if called erroneously.

**CPython implementation detail:** In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use multiprocessing. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.
This module defines the following functions and objects:

```python
threading.active_count()
threading.activeCount()
```

Return the number of `Thread` objects currently alive. The returned count is equal to the length of the list returned by `enumerate()`.

*Changed in version 2.6:* Added `active_count()` spelling.

```python
threading.Condition()
```

A factory function that returns a new condition variable object. A condition variable allows one or more threads to wait until they are notified by another thread.

See Condition Objects.

```python
threading.current_thread()
threading.currentThread()
```

Return the current `Thread` object, corresponding to the caller’s thread of control. If the caller’s thread of control was not created through the `threading` module, a dummy thread object with limited functionality is returned.

*Changed in version 2.6:* Added `current_thread()` spelling.

```python
threading.enumerate()
```

Return a list of all `Thread` objects currently alive. The list includes daemonic threads, dummy thread objects created by `current_thread()`, and the main thread. It excludes terminated threads and threads that have not yet been started.

```python
threading.Event()
```

A factory function that returns a new event object. An event manages a flag that can be set to true with the `set()` method and
reset to false with the `clear()` method. The `wait()` method blocks until the flag is true.

See Event Objects.

class threading. `local`
A class that represents thread-local data. Thread-local data are data whose values are thread specific. To manage thread-local data, just create an instance of `local` (or a subclass) and store attributes on it:

```python
mydata = threading.local()
mydata.x = 1
```

The instance’s values will be different for separate threads.

For more details and extensive examples, see the documentation string of the `_threading_local` module.

*New in version 2.4.*

threading. `Lock()`
A factory function that returns a new primitive lock object. Once a thread has acquired it, subsequent attempts to acquire it block, until it is released; any thread may release it.

See Lock Objects.

threading. `RLock()`
A factory function that returns a new reentrant lock object. A reentrant lock must be released by the thread that acquired it. Once a thread has acquired a reentrant lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

See RLock Objects.
threading.\texttt{Semaphore([value])}

A factory function that returns a new semaphore object. A semaphore manages a counter representing the number of \texttt{release()} calls minus the number of \texttt{acquire()} calls, plus an initial value. The \texttt{acquire()} method blocks if necessary until it can return without making the counter negative. If not given, \texttt{value} defaults to 1.

See Semaphore Objects.

threading.\texttt{BoundedSemaphore([value])}

A factory function that returns a new bounded semaphore object. A bounded semaphore checks to make sure its current value doesn't exceed its initial value. If it does, \texttt{ValueError} is raised. In most situations semaphores are used to guard resources with limited capacity. If the semaphore is released too many times it's a sign of a bug. If not given, \texttt{value} defaults to 1.

\texttt{class} threading.\texttt{Thread}

A class that represents a thread of control. This class can be safely subclassed in a limited fashion.

See Thread Objects.

\texttt{class} threading.\texttt{Timer}

A thread that executes a function after a specified interval has passed.

See Timer Objects.

threading.\texttt{settrace(func)}

Set a trace function for all threads started from the \texttt{threading} module. The \texttt{func} will be passed to \texttt{sys.settrace()} for each thread, before its \texttt{run()} method is called.
New in version 2.3.

threading.\texttt{setprofile}(\texttt{func})

Set a profile function for all threads started from the \texttt{threading} module. The \texttt{func} will be passed to \texttt{sys.setprofile()} for each thread, before its \texttt{run()} method is called.

New in version 2.3.

threading.\texttt{stack_size}([\texttt{size}])

Return the thread stack size used when creating new threads. The optional \texttt{size} argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If \texttt{size} is not specified, 0 is used. If changing the thread stack size is unsupported, a \texttt{ThreadError} is raised. If the specified stack size is invalid, a \texttt{ValueError} is raised and the stack size is unmodified. 32kB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size $> 32$kB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4kB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information). Availability: Windows, systems with POSIX threads.

New in version 2.5.

\texttt{exception} \texttt{threading.ThreadError}

Raised for various threading-related errors as described below. Note that many interfaces use \texttt{RuntimeError} instead of \texttt{ThreadError}. 
Detailed interfaces for the objects are documented below.

The design of this module is loosely based on Java’s threading model. However, where Java makes locks and condition variables basic behavior of every object, they are separate objects in Python. Python’s Thread class supports a subset of the behavior of Java’s Thread class; currently, there are no priorities, no thread groups, and threads cannot be destroyed, stopped, suspended, resumed, or interrupted. The static methods of Java’s Thread class, when implemented, are mapped to module-level functions.

All of the methods described below are executed atomically.
16.2.1. Thread Objects

This class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the run() method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, only override the __init__() and run() methods of this class.

Once a thread object is created, its activity must be started by calling the thread's start() method. This invokes the run() method in a separate thread of control.

Once the thread's activity is started, the thread is considered 'alive'. It stops being alive when its run() method terminates – either normally, or by raising an unhandled exception. The is_alive() method tests whether the thread is alive.

Other threads can call a thread's join() method. This blocks the calling thread until the thread whose join() method is called is terminated.

A thread has a name. The name can be passed to the constructor, and read or changed through the name attribute.

A thread can be flagged as a “daemon thread”. The significance of this flag is that the entire Python program exits when only daemon threads are left. The initial value is inherited from the creating thread. The flag can be set through the daemon property.

Note: Daemon threads are abruptly stopped at shutdown. Their resources (such as open files, database transactions, etc.) may not be released properly. If you want your threads to stop gracefully,
make them non-daemonic and use a suitable signalling mechanism such as an Event.

There is a “main thread” object; this corresponds to the initial thread of control in the Python program. It is not a daemon thread.

There is the possibility that “dummy thread objects” are created. These are thread objects corresponding to “alien threads”, which are threads of control started outside the threading module, such as directly from C code. Dummy thread objects have limited functionality; they are always considered alive and daemonic, and cannot be join()ed. They are never deleted, since it is impossible to detect the termination of alien threads.

class threading.Thread(group=None, target=None, name=None, args=(), kwargs={})

This constructor should always be called with keyword arguments. Arguments are:

  group should be None; reserved for future extension when a ThreadGroup class is implemented.

  target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

  name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.

  args is the argument tuple for the target invocation. Defaults to ()

  kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.
If the subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

**start()**

Start the thread’s activity.

It must be called at most once per thread object. It arranges for the object’s `run()` method to be invoked in a separate thread of control.

This method will raise a `RuntimeError` if called more than once on the same thread object.

**run()**

Method representing the thread’s activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object’s constructor as the `target` argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

**join([timeout])**

Wait until the thread terminates. This blocks the calling thread until the thread whose `join()` method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs.

When the `timeout` argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As `join()` always returns `None`, you must call `isAlive()` after `join()` to decide whether a timeout happened – if the thread is still alive, the
join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

name
A string used for identification purposes only. It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

New in version 2.6.

getName()
setName()
Pre-2.6 API for name.

ident
The ‘thread identifier’ of this thread or None if the thread has not been started. This is a nonzero integer. See the thread.get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

New in version 2.6.

is_alive()
isAlive()
Return whether the thread is alive.
This method returns `True` just before the `run()` method starts until just after the `run()` method terminates. The module function `enumerate()` returns a list of all alive threads.

*Changed in version 2.6:* Added `is_alive()` spelling.

**daemon**

A boolean value indicating whether this thread is a daemon thread (`True`) or not (`False`). This must be set before `start()` is called, otherwise `RuntimeError` is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to `daemon = False`.

The entire Python program exits when no alive non-daemon threads are left.

*New in version 2.6.*

**isDaemon()**

**setDaemon()**

Pre-2.6 API for `daemon`. 
16.2.2. Lock Objects

A primitive lock is a synchronization primitive that is not owned by a particular thread when locked. In Python, it is currently the lowest level synchronization primitive available, implemented directly by the thread extension module.

A primitive lock is in one of two states, “locked” or “unlocked”. It is created in the unlocked state. It has two basic methods, acquire() and release(). When the state is unlocked, acquire() changes the state to locked and returns immediately. When the state is locked, acquire() blocks until a call to release() in another thread changes it to unlocked, then the acquire() call resets it to locked and returns. The release() method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a ThreadError will be raised.

When more than one thread is blocked in acquire() waiting for the state to turn to unlocked, only one thread proceeds when a release() call resets the state to unlocked; which one of the waiting threads proceeds is not defined, and may vary across implementations.

All methods are executed atomically.

Lock.acquire([blocking])

Acquire a lock, blocking or non-blocking.

When invoked with the blocking argument set to True (the default), block until the lock is unlocked, then set it to locked and return True.
When invoked with the *blocking* argument set to *False*, do not block. If a call with *blocking* set to *True* would block, return *False* immediately; otherwise, set the lock to locked and return *True*.

**Lock**. release()

Release a lock.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

When invoked on an unlocked lock, a *ThreadError* is raised.

There is no return value.
16.2.3. RLock Objects

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it uses the concepts of “owning thread” and “recursion level” in addition to the locked/unlocked state used by primitive locks. In the locked state, some thread owns the lock; in the unlocked state, no thread owns it.

To lock the lock, a thread calls its acquire() method; this returns once the thread owns the lock. To unlock the lock, a thread calls its release() method. acquire()/release() call pairs may be nested; only the final release() (the release() of the outermost pair) resets the lock to unlocked and allows another thread blocked in acquire() to proceed.

RLock.acquire([blocking=1])

Acquire a lock, blocking or non-blocking.

When invoked without arguments: if this thread already owns the lock, increment the recursion level by one, and return immediately. Otherwise, if another thread owns the lock, block until the lock is unlocked. Once the lock is unlocked (not owned by any thread), then grab ownership, set the recursion level to one, and return. If more than one thread is blocked waiting until the lock is unlocked, only one at a time will be able to grab ownership of the lock. There is no return value in this case.

When invoked with the blocking argument set to true, do the same thing as when called without arguments, and return true.

When invoked with the blocking argument set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without
arguments, and return true.

RLock.release()

Release a lock, decrementing the recursion level. If after the decrement it is zero, reset the lock to unlocked (not owned by any thread), and if any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling thread.

Only call this method when the calling thread owns the lock. A RuntimeError is raised if this method is called when the lock is unlocked.

There is no return value.
A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default. (Passing one in is useful when several condition variables must share the same lock.)

A condition variable has `acquire()` and `release()` methods that call the corresponding methods of the associated lock. It also has a `wait()` method, and `notify()` and `notifyAll()` methods. These three must only be called when the calling thread has acquired the lock, otherwise a `RuntimeError` is raised.

The `wait()` method releases the lock, and then blocks until it is awakened by a `notify()` or `notifyAll()` call for the same condition variable in another thread. Once awakened, it re-acquires the lock and returns. It is also possible to specify a timeout.

The `notify()` method wakes up one of the threads waiting for the condition variable, if any are waiting. The `notifyAll()` method wakes up all threads waiting for the condition variable.

Note: the `notify()` and `notifyAll()` methods don’t release the lock; this means that the thread or threads awakened will not return from their `wait()` call immediately, but only when the thread that called `notify()` or `notifyAll()` finally relinquishes ownership of the lock.

Tip: the typical programming style using condition variables uses the lock to synchronize access to some shared state; threads that are interested in a particular change of state call `wait()` repeatedly until they see the desired state, while threads that modify the state call `notify()` or `notifyAll()` when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer-consumer
situation with unlimited buffer capacity:

```python
# Consume one item
cv.acquire()
while not an_item_is_available():
    cv.wait()
get_an_available_item()
cv.release()

# Produce one item
cv.acquire()
make_an_item_available()
cv.notify()
cv.release()
```

To choose between `notify()` and `notifyAll()`, consider whether one state change can be interesting for only one or several waiting threads. E.g. in a typical producer-consumer situation, adding one item to the buffer only needs to wake up one consumer thread.

```python
class threading.Condition([lock])
    If the lock argument is given and not None, it must be a Lock or RLock object, and it is used as the underlying lock. Otherwise, a new RLock object is created and used as the underlying lock.
    acquire(*args)
        Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.
    release()
        Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.
    wait([timeout])
        Wait until notified or until a timeout occurs. If the calling thread
```
has not acquired the lock when this method is called, a

**RuntimeError** is raised.

This method releases the underlying lock, and then blocks
until it is awakened by a `notify()` or `notifyAll()` call for the
same condition variable in another thread, or until the optional
timeout occurs. Once awakened or timed out, it re-acquires
the lock and returns.

When the `timeout` argument is present and not `None`, it should
be a floating point number specifying a timeout for the
operation in seconds (or fractions thereof).

When the underlying lock is an `RLock`, it is not released using
its `release()` method, since this may not actually unlock the
lock when it was acquired multiple times recursively. Instead,
an internal interface of the `RLock` class is used, which really
unlocks it even when it has been recursively acquired several
times. Another internal interface is then used to restore the
recursion level when the lock is reacquired.

**notify(n=1)**

By default, wake up one thread waiting on this condition, if
any. If the calling thread has not acquired the lock when this
method is called, a **RuntimeError** is raised.

This method wakes up at most `n` of the threads waiting for the
condition variable; it is a no-op if no threads are waiting.

The current implementation wakes up exactly `n` threads, if at
least `n` threads are waiting. However, it’s not safe to rely on
this behavior. A future, optimized implementation may
occasionally wake up more than `n` threads.

Note: an awakened thread does not actually return from its
\texttt{wait()} call until it can reacquire the lock. Since \texttt{notify()} does not release the lock, its caller should.

\texttt{notify\_all()}
\texttt{notifyAll()}

Wake up all threads waiting on this condition. This method acts like \texttt{notify()}, but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a \texttt{RuntimeError} is raised.

\textit{Changed in version 2.6:} Added \texttt{notify\_all()} spelling.
16.2.5. Semaphore Objects

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra (he used \texttt{P()} and \texttt{V()} instead of \texttt{acquire()} and \texttt{release()}).

A semaphore manages an internal counter which is decremented by each \texttt{acquire()} call and incremented by each \texttt{release()} call. The counter can never go below zero; when \texttt{acquire()} finds that it is zero, it blocks, waiting until some other thread calls \texttt{release()}.

```python
class threading.Semaphore([value])
```

The optional argument gives the initial \textit{value} for the internal counter; it defaults to \texttt{1}. If the \textit{value} given is less than 0, \texttt{ValueError} is raised.

```python
acquire([blocking])
```

Acquire a semaphore.

When invoked without arguments: if the internal counter is larger than zero on entry, decrement it by one and return immediately. If it is zero on entry, block, waiting until some other thread has called \texttt{release()} to make it larger than zero. This is done with proper interlocking so that if multiple \texttt{acquire()} calls are blocked, \texttt{release()} will wake exactly one of them up. The implementation may pick one at random, so the order in which blocked threads are awakened should not be relied on. There is no return value in this case.

When invoked with \textit{blocking} set to true, do the same thing as when called without arguments, and return true.
When invoked with `blocking` set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

`release()`
Release a semaphore, incrementing the internal counter by one. When it was zero on entry and another thread is waiting for it to become larger than zero again, wake up that thread.

### 16.2.5.1. Semaphore Example

Semaphores are often used to guard resources with limited capacity, for example, a database server. In any situation where the size of the resource is fixed, you should use a bounded semaphore. Before spawning any worker threads, your main thread would initialize the semaphore:

```python
maxconnections = 5
...
pool_sema = BoundedSemaphore(value=maxconnections)
```

Once spawned, worker threads call the semaphore's acquire and release methods when they need to connect to the server:

```python
pool_sema.acquire()
conn = connectdb()
... use connection ...
conn.close()
pool_sema.release()
```

The use of a bounded semaphore reduces the chance that a programming error which causes the semaphore to be released more than it's acquired will go undetected.
16.2.6. Event Objects

This is one of the simplest mechanisms for communication between threads: one thread signals an event and other threads wait for it.

An event object manages an internal flag that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true.

class threading.Event

The internal flag is initially false.

`is_set()`
`isSet()`

Return true if and only if the internal flag is true.

*Changed in version 2.6:* Added `is_set()` spelling.

`set()`

Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call `wait()` once the flag is true will not block at all.

`clear()`

Reset the internal flag to false. Subsequently, threads calling `wait()` will block until `set()` is called to set the internal flag to true again.

`wait([timeout])`

Block until the internal flag is true. If the internal flag is true on entry, return immediately. Otherwise, block until another thread calls `set()` to set the flag to true, or until the optional timeout occurs.
When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

This method returns the internal flag on exit, so it will always return True except if a timeout is given and the operation times out.

*Changed in version 2.7:* Previously, the method always returned None.
16.2.7. Timer Objects

This class represents an action that should be run only after a certain amount of time has passed — a timer. Timer is a subclass of Thread and as such also functions as an example of creating custom threads.

Timers are started, as with threads, by calling their start() method. The timer can be stopped (before its action has begun) by calling the cancel() method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

For example:

```python
def hello():
    print "hello, world"

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

class threading.Timer(interval, function, args=[], kwargs={})
    Create a timer that will run function with arguments args and keyword arguments kwargs, after interval seconds have passed.

    cancel()
    Stop the timer, and cancel the execution of the timer's action. This will only work if the timer is still in its waiting stage.
16.2.8. Using locks, conditions, and semaphores in the `with` statement

All of the objects provided by this module that have `acquire()` and `release()` methods can be used as context managers for a `with` statement. The `acquire()` method will be called when the block is entered, and `release()` will be called when the block is exited.

Currently, `Lock`, `RLock`, `Condition`, `Semaphore`, and `BoundedSemaphore` objects may be used as `with` statement context managers. For example:

```python
import threading

some_rlock = threading.RLock()

with some_rlock:
    print "some_rlock is locked while this executes"
```
16.2.9. Importing in threaded code

While the import machinery is thread-safe, there are two key restrictions on threaded imports due to inherent limitations in the way that thread-safety is provided:

- Firstly, other than in the main module, an import should not have the side effect of spawning a new thread and then waiting for that thread in any way. Failing to abide by this restriction can lead to a deadlock if the spawned thread directly or indirectly attempts to import a module.
- Secondly, all import attempts must be completed before the interpreter starts shutting itself down. This can be most easily achieved by only performing imports from non-daemon threads created through the threading module. Daemon threads and threads created directly with the thread module will require some other form of synchronization to ensure they do not attempt imports after system shutdown has commenced. Failure to abide by this restriction will lead to intermittent exceptions and crashes during interpreter shutdown (as the late imports attempt to access machinery which is no longer in a valid state).
16. Optional Operating System Services
16.3. thread — Multiple threads of control

**Note:** The `thread` module has been renamed to `_thread` in Python 3. The `2to3` tool will automatically adapt imports when converting your sources to Python 3; however, you should consider using the high-level `threading` module instead.

This module provides low-level primitives for working with multiple threads (also called *light-weight processes* or *tasks*) — multiple threads of control sharing their global data space. For synchronization, simple locks (also called *mutexes* or *binary semaphores*) are provided. The `threading` module provides an easier to use and higher-level threading API built on top of this module.

The module is optional. It is supported on Windows, Linux, SGI IRIX, Solaris 2.x, as well as on systems that have a POSIX thread (a.k.a. “pthread”) implementation. For systems lacking the `thread` module, the `dummy_thread` module is available. It duplicates this module’s interface and can be used as a drop-in replacement.

It defines the following constant and functions:

- `exception thread.error`
  
  Raised on thread-specific errors.

- `thread.LockType`
  
  This is the type of lock objects.

- `thread.start_new_thread(function, args[, kwargs])`
  
  Start a new thread and return its identifier. The thread executes the function `function` with the argument list `args` (which must be a
tuple). The optional keyword argument specifies a dictionary of keyword arguments. When the function returns, the thread silently exits. When the function terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).

```python
tuple).
```

```python
The optional `kwargs` argument specifies a dictionary of keyword arguments. When the function returns, the thread silently exits. When the function terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).
```

```python
thread.interrupt_main()

Raise a `KeyboardInterrupt` exception in the main thread. A subthread can use this function to interrupt the main thread.
```

```python
New in version 2.3.
```

```python
thread.exit()

Raise the `SystemExit` exception. When not caught, this will cause the thread to exit silently.
```

```python
thread.allocate_lock()

Return a new lock object. Methods of locks are described below. The lock is initially unlocked.
```

```python
thread.get_ident()

Return the ‘thread identifier’ of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.
```

```python
thread.stack_size([size])

Return the thread stack size used when creating new threads. The optional size argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32kB). If size is not specified, 0 is used. If changing the thread stack size is unsupported, the error exception is raised. If the specified stack size is invalid, a `ValueError` is raised and the
```
stack size is unmodified. 32kB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32kB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4kB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information). Availability: Windows, systems with POSIX threads.

*New in version 2.5.*

Lock objects have the following methods:

**lock.**\texttt{acquire}([\texttt{waitflag}])

Without the optional argument, this method acquires the lock unconditionally, if necessary waiting until it is released by another thread (only one thread at a time can acquire a lock — that’s their reason for existence). If the integer \texttt{waitflag} argument is present, the action depends on its value: if it is zero, the lock is only acquired if it can be acquired immediately without waiting, while if it is nonzero, the lock is acquired unconditionally as before. The return value is \texttt{True} if the lock is acquired successfully, \texttt{False} if not.

**lock.**\texttt{release}()

Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

**lock.**\texttt{locked}()

Return the status of the lock: \texttt{True} if it has been acquired by some thread, \texttt{False} if not.

In addition to these methods, lock objects can also be used via the
with statement, e.g.:

```python
import thread

a_lock = thread.allocate_lock()

with a_lock:
    print "a_lock is locked while this executes"
```

Caveats:

- Threads interact strangely with interrupts: the `KeyboardInterrupt` exception will be received by an arbitrary thread. (When the `signal` module is available, interrupts always go to the main thread.)

- Calling `sys.exit()` or raising the `SystemExit` exception is equivalent to calling `thread.exit()`.

- It is not possible to interrupt the `acquire()` method on a lock — the `KeyboardInterrupt` exception will happen after the lock has been acquired.

- When the main thread exits, it is system defined whether the other threads survive. On SGI IRIX using the native thread implementation, they survive. On most other systems, they are killed without executing `try ... finally` clauses or executing object destructors.

- When the main thread exits, it does not do any of its usual cleanup (except that `try ... finally` clauses are honored), and the standard I/O files are not flushed.
16.4. dummy_threading — Drop-in replacement for the threading module

Source code: Lib/dummy_threading.py

This module provides a duplicate interface to the threading module. It is meant to be imported when the thread module is not provided on a platform.

Suggested usage is:

```python
try:
    import threading as _threading
except ImportError:
    import dummy_threading as _threading
```

Be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.
16.5. **dummy_thread** — Drop-in replacement for the **thread** module

**Note:** The **dummy_thread** module has been renamed to **_dummy_thread** in Python 3. The **2to3** tool will automatically adapt imports when converting your sources to Python 3; however, you should consider using the high-lever **dummy_threading** module instead.

**Source code:** Lib/dummy_thread.py

This module provides a duplicate interface to the **thread** module. It is meant to be imported when the **thread** module is not provided on a platform.

Suggested usage is:

```python
try:
    import thread as _thread
except ImportError:
    import dummy_thread as _thread
```

Be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.
16.6. multiprocessing — Process-based “threading” interface

New in version 2.6.
16.6.1. Introduction

The `multiprocessing` module supports spawning processes using an API similar to the `threading` module. The `multiprocessing` package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads. Due to this, the `multiprocessing` module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows.

The `multiprocessing` module also introduces APIs which do not have analogs in the `threading` module. A prime example of this is the `Pool` object which offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data parallelism). The following example demonstrates the common practice of defining such functions in a module so that child processes can successfully import that module. This basic example of data parallelism using `Pool`,

```python
from multiprocessing import Pool

def f(x):
    return x**x

if __name__ == '__main__':
    p = Pool(5)
    print(p.map(f, [1, 2, 3]))
```

will print to standard output

`[1, 4, 9]`

16.6.1.1. The `Process` class
In **multiprocessing**, processes are spawned by creating a `Process` object and then calling its `start()` method. `Process` follows the API of `threading.Thread`. A trivial example of a multiprocess program is

```python
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

To show the individual process IDs involved, here is an expanded example:

```python
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    if hasattr(os, 'getppid'):
        # only available on Unix
        print('parent process:', os.getppid())
        print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

For an explanation of why (on Windows) the `if __name__ == '__main__':` part is necessary, see Programming guidelines.

**16.6.1.2. Exchanging objects between processes**
multiprocessing supports two types of communication channel between processes:

**Queues**

The `Queue` class is a near clone of `Queue.Queue`. For example:

```python
def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print q.get()  # prints '[42, None, 'hello']'
    p.join()
```

Queues are thread and process safe.

**Pipes**

The `Pipe()` function returns a pair of connection objects connected by a pipe which by default is duplex (two-way). For example:

```python
def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print parent_conn.recv()  # prints '[42, None, 'hello']'
    p.join()
```
The two connection objects returned by `Pipe()` represent the two ends of the pipe. Each connection object has `send()` and `recv()` methods (among others). Note that data in a pipe may become corrupted if two processes (or threads) try to read from or write to the *same* end of the pipe at the same time. Of course there is no risk of corruption from processes using different ends of the pipe at the same time.

### 16.6.1.3. Synchronization between processes

`multiprocessing` contains equivalents of all the synchronization primitives from `threading`. For instance one can use a lock to ensure that only one process prints to standard output at a time:

```python
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    print 'hello world', i
    l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

Without using the lock output from the different processes is liable to get all mixed up.

### 16.6.1.4. Sharing state between processes

As mentioned above, when doing concurrent programming it is usually best to avoid using shared state as far as possible. This is particularly true when using multiple processes.

However, if you really do need to use some shared data then
`multiprocessing` provides a couple of ways of doing so.

**Shared memory**

Data can be stored in a shared memory map using `Value` or `Array`. For example, the following code

```python
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

will print

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

The 'd' and 'i' arguments used when creating `num` and `arr` are typecodes of the kind used by the `array` module: 'd' indicates a double precision float and 'i' indicates a signed integer. These shared objects will be process and thread-safe.

For more flexibility in using shared memory one can use the `multiprocessing.sharedctypes` module which supports the creation of arbitrary ctypes objects allocated from shared memory.
Server process

A manager object returned by `Manager()` controls a server process which holds Python objects and allows other processes to manipulate them using proxies.

A manager returned by `Manager()` will support types `list`, `dict`, `Namespace`, `Lock`, `RLock`, `Semaphore`, `BoundedSemaphore`, `Condition`, `Event`, `Queue`, `Value` and `Array`. For example,

```python
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    manager = Manager()

    d = manager.dict()
    l = manager.list(range(10))

    p = Process(target=f, args=(d, l))
    p.start()
    p.join()

    print d
    print l
```

will print

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Server process managers are more flexible than using shared memory objects because they can be made to support arbitrary object types. Also, a single manager can be shared by processes on different computers over a network. They are,
however, slower than using shared memory.

### 16.6.1.5. Using a pool of workers

The **Pool** class represents a pool of worker processes. It has methods which allows tasks to be offloaded to the worker processes in a few different ways.

For example:

```python
def f(x):
    return x**x

if __name__ == '__main__':
    pool = Pool(processes=4)  # start 4 worker proc

    # print "[0, 1, 4,..., 81]"
    print pool.map(f, range(10))

    # print same numbers in arbitrary order
    for i in pool.imap_unordered(f, range(10)):
        print i

    # evaluate "f(20)" asynonymously
    res = pool.apply_async(f, (20,))  # runs in *only* one
    print res.get(timeout=1)  # prints "400"

    # evaluate "os.getpid()" asynonymously
    res = pool.apply_async(os.getpid, ())  # runs in *only* one
    print res.get(timeout=1)  # prints the PID of the

    # launching multiple evaluations asynonymously *may* use
    multiple_results = [pool.apply_async(os.getpid, ()) for i in
                       range(10)]
    [print res.get(timeout=1) for res in multiple_results]

    # make a single worker sleep for 10 secs
    res = pool.apply_async(time.sleep, (10,))
    try:
        print res.get(timeout=1)
```

except TimeoutError:
    print "We lacked patience and got a multiprocessing.TimeoutError"

Note that the methods of a pool should only ever be used by the process which created it.

**Note:** Functionality within this package requires that the `__main__` module be importable by the children. This is covered in Programming guidelines however it is worth pointing out here. This means that some examples, such as the `Pool` examples will not work in the interactive interpreter. For example:

```python
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
... >>> p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
  File "<stdin>", line 1, in f
  File "<stdin>", line 5, in <listcomp>
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
```

(If you try this it will actually output three full tracebacks interleaved in a semi-random fashion, and then you may have to stop the master process somehow.)
16.6.2. Reference

The `multiprocessing` package mostly replicates the API of the `threading` module.

16.6.2.1. Process and exceptions

```python
class multiprocessing.Process(group=None, target=None, name=None, args=(), kwargs={})
```

Process objects represent activity that is run in a separate process. The `Process` class has equivalents of all the methods of `threading.Thread`.

The constructor should always be called with keyword arguments. `group` should always be `None`; it exists solely for compatibility with `threading.Thread`. `target` is the callable object to be invoked by the `run()` method. It defaults to `None`, meaning nothing is called. `name` is the process name. By default, a unique name is constructed of the form ‘Process-N₁:N₂:...:Nₖ’ where N₁,N₂,...,Nₖ is a sequence of integers whose length is determined by the generation of the process. `args` is the argument tuple for the target invocation. `kwargs` is a dictionary of keyword arguments for the target invocation. By default, no arguments are passed to `target`.

If a subclass overrides the constructor, it must make sure it invokes the base class constructor (`Process.__init__()`)) before doing anything else to the process.

`run()`

Method representing the process’s activity.

You may override this method in a subclass. The standard
`run()` method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

**start()**
Start the process's activity.

This must be called at most once per process object. It arranges for the object's `run()` method to be invoked in a separate process.

**join([timeout])**
Block the calling thread until the process whose `join()` method is called terminates or until the optional timeout occurs.

If `timeout` is `None` then there is no timeout.

A process can be joined many times.

A process cannot join itself because this would cause a deadlock. It is an error to attempt to join a process before it has been started.

**name**
The process's name.

The name is a string used for identification purposes only. It has no semantics. Multiple processes may be given the same name. The initial name is set by the constructor.

**is_alive()**
Return whether the process is alive.
Roughly, a process object is alive from the moment the `start()` method returns until the child process terminates.

**daemon**

The process's daemon flag, a Boolean value. This must be set before `start()` is called.

The initial value is inherited from the creating process.

When a process exits, it attempts to terminate all of its daemonic child processes.

Note that a daemonic process is not allowed to create child processes. Otherwise a daemonic process would leave its children orphaned if it gets terminated when its parent process exits. Additionally, these are not Unix daemons or services, they are normal processes that will be terminated (and not joined) if non-daemonic processes have exited.

In addition to the `threading.Thread` API, `Process` objects also support the following attributes and methods:

**pid**

Return the process ID. Before the process is spawned, this will be `None`.

**exitcode**

The child’s exit code. This will be `None` if the process has not yet terminated. A negative value `-N` indicates that the child was terminated by signal `N`.

**authkey**

The process's authentication key (a byte string).

When `multiprocessing` is initialized the main process is assigned a random string using `os.urandom()`.
When a Process object is created, it will inherit the authentication key of its parent process, although this may be changed by setting authkey to another byte string.

See Authentication keys.

terminate()

Terminate the process. On Unix this is done using the SIGTERM signal; on Windows TerminateProcess() is used. Note that exit handlers and finally clauses, etc., will not be executed.

Note that descendant processes of the process will not be terminated – they will simply become orphaned.

Warning: If this method is used when the associated process is using a pipe or queue then the pipe or queue is liable to become corrupted and may become unusable by other process. Similarly, if the process has acquired a lock or semaphore etc. then terminating it is liable to cause other processes to deadlock.

Note that the start(), join(), is_alive(), terminate() and exitcode methods should only be called by the process that created the process object.

Example usage of some of the methods of Process:

```python
>>> import multiprocessing, time, signal
>>> p = multiprocessing.Process(target=time.sleep, args=(100
>>> print p, p.is_alive()  # False
>>> p.start()
>>> print p, p.is_alive()  # True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print p, p.is_alive()  # False
```
exception multiprocessing.BufferTooShort
Exception raised by `Connection.recv_bytes_into()` when the supplied buffer object is too small for the message read.

If `e` is an instance of `BufferTooShort` then `e.args[0]` will give the message as a byte string.

16.6.2.2. Pipes and Queues

When using multiple processes, one generally uses message passing for communication between processes and avoids having to use any synchronization primitives like locks.

For passing messages one can use `Pipe()` (for a connection between two processes) or a queue (which allows multiple producers and consumers).

The `Queue`, `multiprocessing.queues.SimpleQueue` and `JoinableQueue` types are multi-producer, multi-consumer FIFO queues modelled on the `Queue.Queue` class in the standard library. They differ in that `Queue` lacks the `task_done()` and `join()` methods introduced into Python 2.5's `Queue.Queue` class.

If you use `JoinableQueue` then you **must** call `JoinableQueue.task_done()` for each task removed from the queue or else the semaphore used to count the number of unfinished tasks may eventually overflow, raising an exception.

Note that one can also create a shared queue by using a manager object – see `Managers`. 
Note: **multiprocessing** uses the usual **Queue.Empty** and **Queue.Full** exceptions to signal a timeout. They are not available in the **multiprocessing** namespace so you need to import them from **Queue**.

Note: When an object is put on a queue, the object is pickled and a background thread later flushes the pickled data to an underlying pipe. This has some consequences which are a little surprising, but should not cause any practical difficulties – if they really bother you then you can instead use a queue created with a **manager**.

1. After putting an object on an empty queue there may be an infinitesimal delay before the queue's **empty()** method returns **False** and **get_nowait()** can return without raising **Queue.Empty**.
2. If multiple processes are enqueuing objects, it is possible for the objects to be received at the other end out-of-order. However, objects enqueued by the same process will always be in the expected order with respect to each other.

**Warning:** If a process is killed using **Process.terminate()** or **os.kill()** while it is trying to use a **Queue**, then the data in the queue is likely to become corrupted. This may cause any other process to get an exception when it tries to use the queue later on.

**Warning:** As mentioned above, if a child process has put items on a queue (and it has not used **JoinableQueue.cancel_join_thread**), then that process will not terminate until all buffered items have been flushed to the pipe.

This means that if you try joining that process you may get a deadlock unless you are sure that all items which have been put on the queue have been consumed. Similarly, if the child process
is non-daemonic then the parent process may hang on exit when it tries to join all its non-daemonic children.

Note that a queue created using a manager does not have this issue. See Programming guidelines.

For an example of the usage of queues for interprocess communication see Examples.

```
multiprocessing.Pipe([duplex])
```

Returns a pair (conn1, conn2) of Connection objects representing the ends of a pipe.

If duplex is True (the default) then the pipe is bidirectional. If duplex is False then the pipe is unidirectional: conn1 can only be used for receiving messages and conn2 can only be used for sending messages.

```
class multiprocessing.Queue([maxsize])
```

Returns a process shared queue implemented using a pipe and a few locks/semaphores. When a process first puts an item on the queue a feeder thread is started which transfers objects from a buffer into the pipe.

The usual Queue.Empty and Queue.Full exceptions from the standard library's Queue module are raised to signal timeouts.

Queue implements all the methods of Queue.Queue except for task_done() and join().

```
qsize()
```

Return the approximate size of the queue. Because of multithreading/multiprocessing semantics, this number is not reliable.
Note that this may raise \texttt{NotImplementedError} on Unix platforms like Mac OS X where \texttt{sem_getvalue()} is not implemented.

\texttt{empty()}

Return \texttt{True} if the queue is empty, \texttt{False} otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.

\texttt{full()}

Return \texttt{True} if the queue is full, \texttt{False} otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.

\texttt{put(obj[, block[, timeout]])}

Put \texttt{obj} into the queue. If the optional argument \texttt{block} is \texttt{True} (the default) and \texttt{timeout} is \texttt{None} (the default), block if necessary until a free slot is available. If \texttt{timeout} is a positive number, it blocks at most \texttt{timeout} seconds and raises the \texttt{Queue.Full} exception if no free slot was available within that time. Otherwise (\texttt{block} is \texttt{False}), put an item on the queue if a free slot is immediately available, else raise the \texttt{Queue.Full} exception (\texttt{timeout} is ignored in that case).

\texttt{put_nowait(obj)}

Equivalent to \texttt{put(obj, False)}.

\texttt{get([block[, timeout]])}

Remove and return an item from the queue. If optional args \texttt{block} is \texttt{True} (the default) and \texttt{timeout} is \texttt{None} (the default), block if necessary until an item is available. If \texttt{timeout} is a positive number, it blocks at most \texttt{timeout} seconds and raises the \texttt{Queue.Empty} exception if no item was available within that time. Otherwise (\texttt{block} is \texttt{False}), return an item if one is
immediately available, else raise the Queue.Empty exception 
(*timeout* is ignored in that case).

**get_nowait()**
Equivalent to *get(False)*.

Queue has a few additional methods not found in Queue.Queue. These methods are usually unnecessary for most code:

**close()**
Indicate that no more data will be put on this queue by the current process. The background thread will quit once it has flushed all buffered data to the pipe. This is called automatically when the queue is garbage collected.

**join_thread()**
Join the background thread. This can only be used after *close()* has been called. It blocks until the background thread exits, ensuring that all data in the buffer has been flushed to the pipe.

By default if a process is not the creator of the queue then on exit it will attempt to join the queue’s background thread. The process can call *cancel_join_thread()* to make *join_thread()* do nothing.

**cancel_join_thread()**
Prevent *join_thread()* from blocking. In particular, this prevents the background thread from being joined automatically when the process exits – see *join_thread()*.

A better name for this method might be *allow_exit_without_flush()*: It is likely to cause enqueued data to lost, and you almost certainly will not need to use it. It
is really only there if you need the current process to exit immediately without waiting to flush enqueued data to the underlying pipe, and you don’t care about lost data.

**Note:** This class’s functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the functionality in this class will be disabled, and attempts to instantiate a `Queue` will result in an `ImportError`. See issue 3770 for additional information. The same holds true for any of the specialized queue types listed below.

```python
class multiprocessing.queues.SimpleQueue

It is a simplified `Queue` type, very close to a locked `Pipe`.

empty()
Return `True` if the queue is empty, `False` otherwise.

get()
Remove and return an item from the queue.

put(item)
Put `item` into the queue.
```

```python
class multiprocessing.JoinableQueue([maxsize])

`JoinableQueue`, a `Queue` subclass, is a queue which additionally has `task_done()` and `join()` methods.

task_done()
Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each `get()` used to fetch a task, a subsequent call to `task_done()` tells the queue that the processing on the task is complete.

If a `join()` is currently blocking, it will resume when all items
have been processed (meaning that a `task_done()` call was received for every item that had been `put()` into the queue).

Raises a `ValueError` if called more times than there were items placed in the queue.

`join()`
Block until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls `task_done()` to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, `join()` unblocks.

### 16.6.2.3. Miscellaneous

- **multiprocessing.**`active_children()`

  Return list of all live children of the current process.

  Calling this has the side effect of “joining” any processes which have already finished.

- **multiprocessing.**`cpu_count()`

  Return the number of CPUs in the system. May raise `NotImplementedError`.

- **multiprocessing.**`current_process()`

  Return the `Process` object corresponding to the current process.

  An analogue of `threading.current_thread()`.

- **multiprocessing.**`freeze_support()`

  Add support for when a program which uses `multiprocessing`
has been frozen to produce a Windows executable. (Has been tested with py2exe, PyInstaller and cx_Freeze.)

One needs to call this function straight after the `if __name__ == '__main__'` line of the main module. For example:

```python
from multiprocessing import Process, freeze_support

def f():
    print 'hello world!'

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

If the `freeze_support()` line is omitted then trying to run the frozen executable will raise `RuntimeError`.

Calling `freeze_support()` has no effect when invoked on any operating system other than Windows. In addition, if the module is being run normally by the Python interpreter on Windows (the program has not been frozen), then `freeze_support()` has no effect.

`multiprocessing.set_executable()`
Sets the path of the Python interpreter to use when starting a child process. (By default `sys.executable` is used). Embedders will probably need to do some thing like

```python
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

before they can create child processes. (Windows only)

**Note:** `multiprocessing` contains no analogues of `threading.active_count()`, `threading.enumerate()`, `threading.settrace()`, `threading.setprofile()`, `threading.Timer`,
Connection objects allow the sending and receiving of picklable objects or strings. They can be thought of as message oriented connected sockets.

Connection objects are usually created using `Pipe()` – see also Listeners and Clients.

```python
class multiprocessing.Connection

    send(obj)
    Send an object to the other end of the connection which should be read using `recv()`.

    The object must be picklable. Very large pickles (approximately 32 MB+, though it depends on the OS) may raise a `ValueError` exception.

    recv()
    Return an object sent from the other end of the connection using `send()`. Blocks until there its something to receive. Raises `EOFError` if there is nothing left to receive and the other end was closed.

    fileno()
    Return the file descriptor or handle used by the connection.

    close()
    Close the connection.

    This is called automatically when the connection is garbage collected.
```
poll([timeout])
Return whether there is any data available to be read.

If timeout is not specified then it will return immediately. If timeout is a number then this specifies the maximum time in seconds to block. If timeout is None then an infinite timeout is used.

send_bytes(buffer[, offset[, size]])
Send byte data from an object supporting the buffer interface as a complete message.

If offset is given then data is read from that position in buffer. If size is given then that many bytes will be read from buffer. Very large buffers (approximately 32 MB+, though it depends on the OS) may raise a ValueError exception

recv_bytes([maxlength])
Return a complete message of byte data sent from the other end of the connection as a string. Blocks until there is something to receive. Raises EOFError if there is nothing left to receive and the other end has closed.

If maxlength is specified and the message is longer than maxlength then IOError is raised and the connection will no longer be readable.

recv_bytes_into(buffer[, offset])
Read into buffer a complete message of byte data sent from the other end of the connection and return the number of bytes in the message. Blocks until there is something to receive. Raises EOFError if there is nothing left to receive and the other end was closed.
buffer must be an object satisfying the writable buffer interface. If offset is given then the message will be written into the buffer from that position. Offset must be a non-negative integer less than the length of buffer (in bytes).

If the buffer is too short then a BufferTooShort exception is raised and the complete message is available as e.args[0] where e is the exception instance.

For example:

```python
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
[1, 'hello', None]
>>> a.recv_bytes()  # 'hello'
'thank you'
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

**Warning:** The Connection.recv() method automatically unpickles the data it receives, which can be a security risk unless you can trust the process which sent the message.

Therefore, unless the connection object was produced using Pipe() you should only use the recv() and send() methods after performing some sort of authentication. See Authentication keys.

**Warning:** If a process is killed while it is trying to read or write to a pipe then the data in the pipe is likely to become corrupted,
because it may become impossible to be sure where the message boundaries lie.

16.6.2.5. Synchronization primitives

Generally synchronization primitives are not as necessary in a multiprocess program as they are in a multithreaded program. See the documentation for threading module.

Note that one can also create synchronization primitives by using a manager object – see Managers.

```python
class multiprocessing.BoundedSemaphore([value])
A bounded semaphore object: a close analog of threading.BoundedSemaphore.

A solitary difference from its close analog exists: its acquire method's first argument is named block and it supports an optional second argument timeout, as is consistent with Lock.acquire().

Note: On Mac OS X, this is indistinguishable from Semaphore because sem_getvalue() is not implemented on that platform.
```

```python
class multiprocessing.Condition([lock])

If lock is specified then it should be a Lock or RLock object from multiprocessing.
```

```python
class multiprocessing.Event
A clone of threading.Event. This method returns the state of the internal semaphore on exit, so it will always return True except if
a timeout is given and the operation times out.

*Changed in version 2.7:* Previously, the method always returned None.

class multiprocessing.Lock
A non-recursive lock object: a close analog of threading.Lock. Once a process or thread has acquired a lock, subsequent attempts to acquire it from any process or thread will block until it is released; any process or thread may release it. The concepts and behaviors of threading.Lock as it applies to threads are replicated here in multiprocessing.Lock as it applies to either processes or threads, except as noted.

Note that Lock is actually a factory function which returns an instance of multiprocessing.synchronize.Lock initialized with a default context.

Lock supports the context manager protocol and thus may be used in with statements.

acquire(block=True, timeout=None)
Acquire a lock, blocking or non-blocking.

With the block argument set to True (the default), the method call will block until the lock is in an unlocked state, then set it to locked and return True. Note that the name of this first argument differs from that in threading.Lock.acquire().

With the block argument set to False, the method call does not block. If the lock is currently in a locked state, return False; otherwise set the lock to a locked state and return True.
When invoked with a positive, floating-point value for `timeout`, block for at most the number of seconds specified by `timeout` as long as the lock can not be acquired. Invocations with a negative value for `timeout` are equivalent to a `timeout` of zero. Invocations with a `timeout` value of `None` (the default) set the timeout period to infinite. The `timeout` argument has no practical implications if the `block` argument is set to `False` and is thus ignored. Returns `True` if the lock has been acquired or `False` if the timeout period has elapsed. Note that the `timeout` argument does not exist in this method's analog, `threading.Lock.acquire()`.

**release()**

Release a lock. This can be called from any process or thread, not only the process or thread which originally acquired the lock. Behavior is the same as in `threading.Lock.release()` except that when invoked on an unlocked lock, a `ValueError` is raised.

class multiprocessing.RLock

A recursive lock object: a close analog of `threading.RLock`. A recursive lock must be released by the process or thread that acquired it. Once a process or thread has acquired a recursive lock, the same process or thread may acquire it again without blocking; that process or thread must release it once for each time it has been acquired.

Note that `RLock` is actually a factory function which returns an instance of `multiprocessing.synchronize.RLock` initialized with a default context.

`RLock` supports the `context manager` protocol and thus may be
used in `with` statements.

**acquire**(*block=True, timeout=None*)

Acquire a lock, blocking or non-blocking.

When invoked with the `block` argument set to `True`, block until the lock is in an unlocked state (not owned by any process or thread) unless the lock is already owned by the current process or thread. The current process or thread then takes ownership of the lock (if it does not already have ownership) and the recursion level inside the lock increments by one, resulting in a return value of `True`. Note that there are several differences in this first argument's behavior compared to the implementation of `threading.RLock.acquire()`, starting with the name of the argument itself.

When invoked with the `block` argument set to `False`, do not block. If the lock has already been acquired (and thus is owned) by another process or thread, the current process or thread does not take ownership and the recursion level within the lock is not changed, resulting in a return value of `False`. If the lock is in an unlocked state, the current process or thread takes ownership and the recursion level is incremented, resulting in a return value of `True`.

Use and behaviors of the `timeout` argument are the same as in `Lock.acquire()`. Note that the `timeout` argument does not exist in this method's analog, `threading.RLock.acquire()`.

**release**()

Release a lock, decrementing the recursion level. If after the decrement the recursion level is zero, reset the lock to unlocked (not owned by any process or thread) and if any other processes or threads are blocked waiting for the lock to
become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling process or thread.

Only call this method when the calling process or thread owns the lock. An :class:`AssertionError` is raised if this method is called by a process or thread other than the owner or if the lock is in an unlocked (unowned) state. Note that the type of exception raised in this situation differs from the implemented behavior in :meth:`threading.RLock.release`

```python
class multiprocessing.Semaphore([value])

A semaphore object: a close analog of :class:`threading.Semaphore`

A solitary difference from its close analog exists: its :meth:`acquire` method's first argument is named `block` and it supports an optional second argument `timeout`, as is consistent with :meth:`Lock.acquire`
```

**Note:** The :meth:`acquire()` method of :class:`BoundedSemaphore`, :class:`Lock`, :class:`RLock` and :class:`Semaphore` has a timeout parameter not supported by the equivalents in :mod:`threading`. The signature is acquire(block=True, timeout=None) with keyword parameters being acceptable. If `block` is True and `timeout` is not None then it specifies a timeout in seconds. If `block` is False then `timeout` is ignored.

On Mac OS X, :func:`sem_timedwait` is unsupported, so calling :meth:`acquire` with a timeout will emulate that function’s behavior using a sleeping loop.

**Note:** If the SIGINT signal generated by `ctrl-c` arrives while the main thread is blocked by a call to :meth:`BoundedSemaphore.acquire`,
Lock.acquire(), RLock.acquire(), Semaphore.acquire(),
Condition.acquire() or Condition.wait() then the call will be
immediately interrupted and KeyboardInterrupt will be raised.

This differs from the behaviour of threading where SIGINT will be
ignored while the equivalent blocking calls are in progress.

**Note:** Some of this package's functionality requires a functioning
shared semaphore implementation on the host operating system.
Without one, the multiprocessing.synchronize module will be
disabled, and attempts to import it will result in an ImportError.
See issue 3770 for additional information.

### 16.6.2.6. Shared ctypes Objects

It is possible to create shared objects using shared memory which
can be inherited by child processes.

```python
multiprocessing.Value(typecode_or_type, *args[, lock])
```

Return a ctypes object allocated from shared memory. By default
the return value is actually a synchronized wrapper for the object.

`typecode_or_type` determines the type of the returned object: it is
either a ctypes type or a one character typecode of the kind used
by the array module. `*args` is passed on to the constructor for the
type.

If `lock` is `True` (the default) then a new recursive lock object is
created to synchronize access to the value. If `lock` is a Lock or
RLock object then that will be used to synchronize access to the
value. If `lock` is `False` then access to the returned object will not
be automatically protected by a lock, so it will not necessarily be
“process-safe”. 
Operations like `+=` which involve a read and write are not atomic. So if, for instance, you want to atomically increment a shared value it is insufficient to just do

```
counter.value += 1
```

Assuming the associated lock is recursive (which it is by default) you can instead do

```
with counter.get_lock():
    counter.value += 1
```

Note that `lock` is a keyword-only argument.

`multiprocessing.Array(typecode_or_type, size_or_initializer, *, lock=True)`

Return a ctypes array allocated from shared memory. By default the return value is actually a synchronized wrapper for the array.

`typecode_or_type` determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the `array` module. If `size_or_initializer` is an integer, then it determines the length of the array, and the array will be initially zeroed. Otherwise, `size_or_initializer` is a sequence which is used to initialize the array and whose length determines the length of the array.

If `lock` is `True` (the default) then a new lock object is created to synchronize access to the value. If `lock` is a `Lock` or `RLock` object then that will be used to synchronize access to the value. If `lock` is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that `lock` is a keyword only argument.
Note that an array of `ctypes.c_char` has `value` and `raw` attributes which allow one to use it to store and retrieve strings.

### 16.6.2.6.1. The `multiprocessing.sharedctypes` module

The `multiprocessing.sharedctypes` module provides functions for allocating `ctypes` objects from shared memory which can be inherited by child processes.

**Note:** Although it is possible to store a pointer in shared memory remember that this will refer to a location in the address space of a specific process. However, the pointer is quite likely to be invalid in the context of a second process and trying to dereference the pointer from the second process may cause a crash.

```python
multiprocessing.sharedctypes.RawArray(typecode_or_type, size_or_initializer)
```

Return a `ctypes` array allocated from shared memory.

`typecode_or_type` determines the type of the elements of the returned array: it is either a `ctypes` type or a one character typecode of the kind used by the `array` module. If `size_or_initializer` is an integer then it determines the length of the array, and the array will be initially zeroed. Otherwise `size_or_initializer` is a sequence which is used to initialize the array and whose length determines the length of the array.

Note that setting and getting an element is potentially non-atomic – use `Array()` instead to make sure that access is automatically synchronized using a lock.

```python
multiprocessing.sharedctypes.RawValue(typecode_or_type, *args)
```

Return a `ctypes` object allocated from shared memory.

`typecode_or_type` determines the type of the returned object: it is
either a ctypes type or a one character typecode of the kind used by the array module. *args is passed on to the constructor for the type.

Note that setting and getting the value is potentially non-atomic – use Value() instead to make sure that access is automatically synchronized using a lock.

Note that an array of ctypes.c_char has value and raw attributes which allow one to use it to store and retrieve strings – see documentation for ctypes.

multiprocessing.sharedctypes.Array(typecode_or_type, size_or_initializer, *args[, lock])

The same as RawArray() except that depending on the value of lock a process-safe synchronization wrapper may be returned instead of a raw ctypes array.

If lock is True (the default) then a new lock object is created to synchronize access to the value. If lock is a Lock or RLock object then that will be used to synchronize access to the value. If lock is False then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that lock is a keyword-only argument.

multiprocessing.sharedctypes.Value(typecode_or_type, *args[, lock])

The same as RawValue() except that depending on the value of lock a process-safe synchronization wrapper may be returned instead of a raw ctypes object.

If lock is True (the default) then a new lock object is created to
synchronize access to the value. If `lock` is a `Lock` or `RLock` object then that will be used to synchronize access to the value. If `lock` is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that `lock` is a keyword-only argument.

```python
multiprocessing.sharedctypes.copy(obj)
```

Return a ctypes object allocated from shared memory which is a copy of the ctypes object `obj`.

```python
multiprocessing.sharedctypes.synchronized(obj, lock]
```

Return a process-safe wrapper object for a ctypes object which uses `lock` to synchronize access. If `lock` is `None` (the default) then a `multiprocessing.RLock` object is created automatically.

A synchronized wrapper will have two methods in addition to those of the object it wraps: `get_obj()` returns the wrapped object and `get_lock()` returns the lock object used for synchronization.

Note that accessing the ctypes object through the wrapper can be a lot slower than accessing the raw ctypes object.

The table below compares the syntax for creating shared ctypes objects from shared memory with the normal ctypes syntax. (In the table `MyStruct` is some subclass of `ctypes.Structure`.)

<table>
<thead>
<tr>
<th>ctypes</th>
<th>sharedctypes using type</th>
<th>sharedctypes using typecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_double(2.4)</td>
<td>RawValue(c_double, 2.4)</td>
<td>RawValue('d', 2.4)</td>
</tr>
<tr>
<td>MyStruct(4, 6)</td>
<td>RawValue(MyStruct, 4, 6)</td>
<td></td>
</tr>
<tr>
<td>(c_short * 7)()</td>
<td>RawArray(c_short, 7)</td>
<td>RawArray('h', 7)</td>
</tr>
</tbody>
</table>
Below is an example where a number of ctypes objects are modified by a child process:

```python
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', 'hello world', lock=lock)
    A = Array(Point, [(1.875, -6.25), (-5.75, 2.0), (2.375, 9.5)], lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()

    print n.value
    print x.value
    print s.value
    print [(a.x, a.y) for a in A]
```

The results printed are

```
49
0.1111111111111111
HELLO WORLD
```
16.6.2.7. Managers

Managers provide a way to create data which can be shared between different processes. A manager object controls a server process which manages *shared objects*. Other processes can access the shared objects by using proxies.

```python
multiprocessing.Manager()
```

Returns a started *SyncManager* object which can be used for sharing objects between processes. The returned manager object corresponds to a spawned child process and has methods which will create shared objects and return corresponding proxies.

Manager processes will be shutdown as soon as they are garbage collected or their parent process exits. The manager classes are defined in the `multiprocessing.managers` module:

```python
class multiprocessing.managers.BaseManager([address[, authkey]])
```

Create a BaseManager object.

Once created one should call `start()` or `get_server().serve_forever()` to ensure that the manager object refers to a started manager process.

- `address` is the address on which the manager process listens for new connections. If `address` is `None` then an arbitrary one is chosen.

- `authkey` is the authentication key which will be used to check the validity of incoming connections to the server process. If `authkey`
is None then current_process().authkey. Otherwise authkey is used and it must be a string.

start([initializer[, initargs]])

Start a subprocess to start the manager. If initializer is not None then the subprocess will call initializer(*initargs) when it starts.

get_server()

Returns a server object which represents the actual server under the control of the Manager. The Server object supports the serve_forever() method:

```python
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000), authkey='abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

Server additionally has an address attribute.

connect()

Connect a local manager object to a remote manager process:

```python
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 5000), authkey='abc')
>>> m.connect()
```

shutdown()

Stop the process used by the manager. This is only available if start() has been used to start the server process.

This can be called multiple times.
register(typeid[, callable[, proxytype[, exposed[, method_to_typeid[, create_method]]]]])

A classmethod which can be used for registering a type or callable with the manager class.

typeid is a “type identifier” which is used to identify a particular type of shared object. This must be a string.

callable is a callable used for creating objects for this type identifier. If a manager instance will be created using the from_address() classmethod or if the create_method argument is False then this can be left as None.

proxytype is a subclass of BaseProxy which is used to create proxies for shared objects with this typeid. If None then a proxy class is created automatically.

exposed is used to specify a sequence of method names which proxies for this typeid should be allowed to access using BaseProxy._callmethod(). (If exposed is None then proxytype._exposed_ is used instead if it exists.) In the case where no exposed list is specified, all “public methods” of the shared object will be accessible. (Here a “public method” means any attribute which has a __call__() method and whose name does not begin with '_'.)

method_to_typeid is a mapping used to specify the return type of those exposed methods which should return a proxy. It maps method names to typeid strings. (If method_to_typeid is None then proxytype._method_to_typeid_ is used instead if it exists.) If a method's name is not a key of this mapping or if the mapping is None then the object returned by the method will be copied by value.
create_method determines whether a method should be created with name typeid which can be used to tell the server process to create a new shared object and return a proxy for it. By default it is True.

BaseManager instances also have one read-only property:

**address**

The address used by the manager.

class multiprocessing.managers.SyncManager

A subclass of BaseManager which can be used for the synchronization of processes. Objects of this type are returned by multiprocessing.Manager().

It also supports creation of shared lists and dictionaries.

**BoundedSemaphore([value])**

Create a shared threading.BoundedSemaphore object and return a proxy for it.

**Condition([lock])**

Create a shared threading.Condition object and return a proxy for it.

If lock is supplied then it should be a proxy for a threading.Lock or threading.RLock object.

**Event()**

Create a shared threading.Event object and return a proxy for it.

**Lock()**

Create a shared threading.Lock object and return a proxy for
it.

**Namespace()**
Create a shared `Namespace` object and return a proxy for it.

**Queue([maxsize])**
Create a shared `queue.Queue` object and return a proxy for it.

**RLock()**
Create a shared `threading.RLock` object and return a proxy for it.

**Semaphore([value])**
Create a shared `threading.Semaphore` object and return a proxy for it.

**Array(typecode, sequence)**
Create an array and return a proxy for it.

**Value(typecode, value)**
Create an object with a writable `value` attribute and return a proxy for it.

**dict()**
**dict(mapping)**
**dict(sequence)**
Create a shared `dict` object and return a proxy for it.

**list()**
**list(sequence)**
Create a shared `list` object and return a proxy for it.

**Note:** Modifications to mutable values or items in dict and list proxies will not be propagated through the manager, because
the proxy has no way of knowing when its values or items are modified. To modify such an item, you can re-assign the modified object to the container proxy:

```python
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synced, but
# reassigning the dictionary, the proxy is notified of the change
lproxy[0] = d
```

```python
class multiprocessing.managers.Namespace
A type that can register with SyncManager.

A namespace object has no public methods, but does have writable attributes. Its representation shows the values of its attributes.

However, when using a proxy for a namespace object, an attribute beginning with `_` will be an attribute of the proxy and not an attribute of the referent:

```python
>>> manager = multiprocessing.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3  # this is an attribute of the proxy
>>> print Global
Namespace(x=10, y='hello')
```

### 16.6.2.7.1. Customized managers

To create one's own manager, one creates a subclass of BaseManager and uses the register() classmethod to register new
types or callables with the manager class. For example:

```python
from multiprocessing.managers import BaseManager

class MathsClass(object):
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    manager = MyManager()
    manager.start()
    maths = manager.Maths()
    print maths.add(4, 3)  # prints 7
    print maths.mul(7, 8)  # prints 56
```

### 16.6.2.7.2. Using a remote manager

It is possible to run a manager server on one machine and have clients use it from other machines (assuming that the firewalls involved allow it).

Running the following commands creates a server for a single shared queue which remote clients can access:

```python
>>> from multiprocessing.managers import BaseManager
>>> import Queue
>>> queue = Queue.Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager.register('', address=('', 50000), authkey='abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

One client can access the server as follows:
```python
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey='abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')

Another client can also use it:

```python
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey='abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'

Local processes can also access that queue, using the code from above on the client to access it remotely:

```python
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...     super(Worker, self).__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
... >>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey='abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```
16.6.2.8. Proxy Objects

A proxy is an object which refers to a shared object which lives (presumably) in a different process. The shared object is said to be the referent of the proxy. Multiple proxy objects may have the same referent.

A proxy object has methods which invoke corresponding methods of its referent (although not every method of the referent will necessarily be available through the proxy). A proxy can usually be used in most of the same ways that its referent can:

```python
>>> from multiprocessing import Manager
>>> manager = Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print l
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print repr(l)
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

Notice that applying `str()` to a proxy will return the representation of the referent, whereas applying `repr()` will return the representation of the proxy.

An important feature of proxy objects is that they are pickleable so they can be passed between processes. Note, however, that if a proxy is sent to the corresponding manager’s process then unpickling it will produce the referent itself. This means, for example, that one shared object can contain a second:

```python
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)  # referent of a now contains referent of b
>>> print a, b
[a, b]
```
Note: The proxy types in multiprocessing do nothing to support comparisons by value. So, for instance, we have:

```python
>>> manager.list([1, 2, 3]) == [1, 2, 3]
False
```

One should just use a copy of the referent instead when making comparisons.

class multiprocessing.managers.BaseProxy

Proxy objects are instances of subclasses of BaseProxy.

```
def _callmethod(methodname[, args[, kwds]])
    Call and return the result of a method of the proxy's referent.

    If proxy is a proxy whose referent is obj then the expression
    `proxy._callmethod(methodname, args, kwds)`

    will evaluate the expression
    `getattr(obj, methodname)(*args, **kwds)`

    in the manager's process.

    The returned value will be a copy of the result of the call or a
    proxy to a new shared object – see documentation for the
    method_to_typeid argument of BaseManager.register().

    If an exception is raised by the call, then is re-raised by
    _callmethod(). If some other exception is raised in the
```
manager’s process then this is converted into a RemoteError exception and is raised by _callmethod().

Note in particular that an exception will be raised if methodname has not been exposed.

An example of the usage of _callmethod():

```python
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getslice__', (2, 7))  # equiv to `l[2:7]`
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))  # equiv to `l[20]`
Traceback (most recent call last):
... IndexError: list index out of range
```

___getvalue___

Return a copy of the referent.

If the referent is unpicklable then this will raise an exception.

___repr___

Return a representation of the proxy object.

___str___

Return the representation of the referent.

### 16.6.2.8.1. Cleanup

A proxy object uses a weakref callback so that when it gets garbage collected it deregisters itself from the manager which owns its referent.

A shared object gets deleted from the manager process when there are no longer any proxies referring to it.
16.6.2.9. Process Pools

One can create a pool of processes which will carry out tasks submitted to it with the `Pool` class.

```python
class multiprocessing.Pool([processes[, initializer[, initargs[, maxtasksperchild]]]]))
```

A process pool object which controls a pool of worker processes to which jobs can be submitted. It supports asynchronous results with timeouts and callbacks and has a parallel map implementation.

- `processes` is the number of worker processes to use. If `processes` is `None` then the number returned by `cpu_count()` is used. If `initializer` is not `None` then each worker process will call `initializer(*initargs)` when it starts.

Note that the methods of the pool object should only be called by the process which created the pool.

*New in version 2.7:* `maxtasksperchild` is the number of tasks a worker process can complete before it will exit and be replaced with a fresh worker process, to enable unused resources to be freed. The default `maxtasksperchild` is `None`, which means worker processes will live as long as the pool.

**Note:** Worker processes within a `Pool` typically live for the complete duration of the Pool's work queue. A frequent pattern found in other systems (such as Apache, mod_wsgi, etc) to free resources held by workers is to allow a worker within a pool to complete only a set amount of work before being exiting, being cleaned up and a new process spawned to replace the old one. The `maxtasksperchild` argument to the `Pool` exposes this ability.
apply(func[, args[, kwds]])
Equivalent of the apply() built-in function. It blocks until the result is ready, so apply_async() is better suited for performing work in parallel. Additionally, func is only executed in one of the workers of the pool.

apply_async(func[, args[, kwds[, callback]]])
A variant of the apply() method which returns a result object.

If callback is specified then it should be a callable which accepts a single argument. When the result becomes ready callback is applied to it (unless the call failed). callback should complete immediately since otherwise the thread which handles the results will get blocked.

map(func, iterable[, chunksize])
A parallel equivalent of the map() built-in function (it supports only one iterable argument though). It blocks until the result is ready.

This method chops the iterable into a number of chunks which it submits to the process pool as separate tasks. The (approximate) size of these chunks can be specified by setting chunksize to a positive integer.

map_async(func, iterable[, chunksize[, callback]])
A variant of the map() method which returns a result object.

If callback is specified then it should be a callable which accepts a single argument. When the result becomes ready callback is applied to it (unless the call failed). callback should complete immediately since otherwise the thread which
handles the results will get blocked.

**imap(func, iterable[, chunksize])**

An equivalent of `itertools.imap()`.

The `chunksize` argument is the same as the one used by the `map()` method. For very long iterables using a large value for `chunksize` can make the job complete much faster than using the default value of 1.

Also if `chunksize` is 1 then the `next()` method of the iterator returned by the `imap()` method has an optional `timeout` parameter: `next(timeout)` will raise `multiprocessing.TimeoutError` if the result cannot be returned within `timeout` seconds.

**imap_unordered(func, iterable[, chunksize])**

The same as `imap()` except that the ordering of the results from the returned iterator should be considered arbitrary. (Only when there is only one worker process is the order guaranteed to be “correct”.)

**close()**

Prevents any more tasks from being submitted to the pool. Once all the tasks have been completed the worker processes will exit.

**terminate()**

Stops the worker processes immediately without completing outstanding work. When the pool object is garbage collected `terminate()` will be called immediately.

**join()**
Wait for the worker processes to exit. One must call `close()` or `terminate()` before using `join()`.

```python
class multiprocessing.poolAsyncResult
    The class of the result returned by `Pool.apply_async()` and `Pool.map_async()`.

    get([timeout])
    Return the result when it arrives. If `timeout` is not `None` and the result does not arrive within `timeout` seconds then `multiprocessing.TimeoutError` is raised. If the remote call raised an exception then that exception will be reraised by `get()`.

    wait([timeout])
    Wait until the result is available or until `timeout` seconds pass.

    ready()
    Return whether the call has completed.

    successful()
    Return whether the call completed without raising an exception. Will raise `AssertionError` if the result is not ready.
```

The following example demonstrates the use of a pool:

```python
from multiprocessing import Pool
import time

def f(x):  
    return x**x

if __name__ == '__main__':
    pool = Pool(processes=4)  # start 4 worker proc

    result = pool.apply_async(f, (10,))  # evaluate "f(10)" as
    print result.get(timeout=1)  # prints "100" unless
```
```python
print pool.map(f, range(10))  # prints "[0, 1, 4,...

it = pool.imap(f, range(10))
print it.next()  # prints "0"
print it.next()  # prints "1"
print it.next(timeout=1)  # prints "4" unless you...

result = pool.apply_async(time.sleep, (10,))
print result.get(timeout=1)  # raises multiprocessing.TimeoutError
```

16.6.2.10. Listeners and Clients

Usually message passing between processes is done using queues or by using `Connection` objects returned by `Pipe()`.

However, the `multiprocessing.connection` module allows some extra flexibility. It basically gives a high level message oriented API for dealing with sockets or Windows named pipes, and also has support for `digest authentication` using the `hmac` module.

```python
multiprocessing.connection.deliver_challenge(connection, authkey)
```

Send a randomly generated message to the other end of the connection and wait for a reply.

If the reply matches the digest of the message using `authkey` as the key then a welcome message is sent to the other end of the connection. Otherwise `AuthenticationError` is raised.

```python
multiprocessing.connection.answer_challenge(connection, authkey)
```

Receive a message, calculate the digest of the message using `authkey` as the key, and then send the digest back.

If a welcome message is not received, then `AuthenticationError` is raised.
multiprocessing.connection.client\(\textit{address}[, \textit{family}[\textit{authenticate}[, \textit{authkey}]]]\)

Attempt to set up a connection to the listener which is using address \textit{address}, returning a \texttt{Connection}.

The type of the connection is determined by \textit{family} argument, but this can generally be omitted since it can usually be inferred from the format of \textit{address}. (See Address Formats)

If \textit{authenticate} is \texttt{True} or \textit{authkey} is a string then digest authentication is used. The key used for authentication will be either \textit{authkey} or \texttt{current_process().authkey} if \textit{authkey} is \texttt{None}. If authentication fails then \texttt{AuthenticationError} is raised. See Authentication keys.

class multiprocessing.connection.Listener([\textit{address}[\textit{family}[\textit{backlog}[\textit{authenticate}[\textit{authkey}]]]]])

A wrapper for a bound socket or Windows named pipe which is ‘listening’ for connections.

\textit{address} is the address to be used by the bound socket or named pipe of the listener object.

**Note:** If an address of ‘0.0.0.0’ is used, the address will not be a connectable end point on Windows. If you require a connectable end-point, you should use ‘127.0.0.1’.

\textit{family} is the type of socket (or named pipe) to use. This can be one of the strings \texttt{'AF_INET'} (for a TCP socket), \texttt{'AF_UNIX'} (for a Unix domain socket) or \texttt{'AF_PIPE'} (for a Windows named pipe). Of these only the first is guaranteed to be available. If \textit{family} is \texttt{None} then the family is inferred from the format of \textit{address}. If \textit{address} is also \texttt{None} then a default is chosen. This default is the
family which is assumed to be the fastest available. See Address Formats. Note that if family is 'AF_UNIX' and address is None then the socket will be created in a private temporary directory created using tempfile.mkstemp().

If the listener object uses a socket then backlog (1 by default) is passed to the listen() method of the socket once it has been bound.

If authenticate is True (False by default) or authkey is not None then digest authentication is used.

If authkey is a string then it will be used as the authentication key; otherwise it must be None.

If authkey is None and authenticate is True then current_process().authkey is used as the authentication key. If authkey is None and authenticate is False then no authentication is done. If authentication fails then AuthenticationError is raised. See Authentication keys.

accept()

Accept a connection on the bound socket or named pipe of the listener object and return a Connection object. If authentication is attempted and fails, then AuthenticationError is raised.

close()

Close the bound socket or named pipe of the listener object. This is called automatically when the listener is garbage collected. However it is advisable to call it explicitly.

Listener objects have the following read-only properties:

address
The address which is being used by the Listener object.

last_accepted
The address from which the last accepted connection came. If this is unavailable then it is None.

The module defines two exceptions:

exception multiprocessing.connection.AuthenticationError
Exception raised when there is an authentication error.

Examples

The following server code creates a listener which uses 'secret password' as an authentication key. It then waits for a connection and sends some data to the client:

```python
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)  # family is deduced to be 'AF
listener = Listener(address, authkey='secret password')

conn = listener.accept()
print 'connection accepted from', listener.last_accepted

conn.send([2.25, None, 'junk', float])

conn.send_bytes('hello')

conn.send_bytes(array('i', [42, 1729]))

conn.close()
listener.close()
```

The following code connects to the server and receives some data from the server:

```python
from multiprocessing.connection import Client
```
from array import array

address = ('localhost', 6000)
conn = Client(address, authkey='secret password')

print conn.recv()  # => [2.25, None, 'junk', float]
print conn.recv_bytes()  # => 'hello'

arr = array('i', [0, 0, 0, 0, 0])
print conn.recv_bytes_into(arr)  # => 8
print arr  # => array('i', [42, 1729, 0, 0, 0])

conn.close()

16.6.2.10.1. Address Formats

- An 'AF_INET' address is a tuple of the form (hostname, port) where hostname is a string and port is an integer.

- An 'AF_UNIX' address is a string representing a filename on the filesystem.

- An 'AF_PIPE' address is a string of the form r'\\PipeName'. To use Client() to connect to a named pipe on a remote computer called ServerName one should use an address of the form r'\\ServerName\PipeName' instead.

Note that any string beginning with two backslashes is assumed by default to be an 'AF_PIPE' address rather than an 'AF_UNIX' address.

16.6.2.11. Authentication keys

When one uses connection.recv, the data received is automatically unpickled. Unfortunately unpickling data from an untrusted source is
a security risk. Therefore Listener and Client() use the hmac module to provide digest authentication.

An authentication key is a string which can be thought of as a password: once a connection is established both ends will demand proof that the other knows the authentication key. (Demonstrating that both ends are using the same key does not involve sending the key over the connection.)

If authentication is requested but no authentication key is specified then the return value of current_process().authkey is used (see Process). This value will be automatically inherited by any Process object that the current process creates. This means that (by default) all processes of a multi-process program will share a single authentication key which can be used when setting up connections between themselves.

Suitable authentication keys can also be generated by using os.urandom().

16.6.2.12. Logging

Some support for logging is available. Note, however, that the logging package does not use process shared locks so it is possible (depending on the handler type) for messages from different processes to get mixed up.

multiprocessing.get_logger()

Returns the logger used by multiprocessing. If necessary, a new one will be created.

When first created the logger has level logging.NOTSET and no default handler. Messages sent to this logger will not by default propagate to the root logger.
Note that on Windows child processes will only inherit the level of the parent process’s logger – any other customization of the logger will not be inherited.

```
multiprocessing.log_to_stderr()
```

This function performs a call to get_logger() but in addition to returning the logger created by get_logger, it adds a handler which sends output to `sys.stderr` using format `'[%(levelname)s/%(processName)s] %(message)s'`.

Below is an example session with logging turned on:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

In addition to having these two logging functions, the multiprocessing also exposes two additional logging level attributes. These are `SUBWARNING` and `SUBDEBUG`. The table below illustrates where theses fit in the normal level hierarchy.

<table>
<thead>
<tr>
<th>Level</th>
<th>Numeric value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBWARNING</td>
<td>25</td>
</tr>
<tr>
<td>SUBDEBUG</td>
<td>5</td>
</tr>
</tbody>
</table>

For a full table of logging levels, see the `logging` module.

These additional logging levels are used primarily for certain debug messages within the multiprocessing module. Below is the same
example as above, except with `SUBDEBUG` enabled:

```python
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(multiprocessing.SUBDEBUG)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-....] child process calling self.run()
[INFO/SyncManager-....] created temp directory /.../pymp-....
[INFO/SyncManager-....] manager serving at '/.../pymp-djGBXN/lis
>>> del m
[SUBDEBUG/MainProcess] finalizer calling ...
[INFO/MainProcess] sending shutdown message to manager
[DEBUG/SyncManager-....] manager received shutdown message
[SUBDEBUG/SyncManager-....] calling <Finalize object, callback=u
[SUBDEBUG/SyncManager-....] finalizer calling <built-in function
[SUBDEBUG/SyncManager-....] calling <Finalize object, dead>
[SUBDEBUG/SyncManager-....] finalizer calling <function rmtree a
[INFO/SyncManager-....] manager exiting with exitcode 0
```

**16.6.2.13. The `multiprocessing.dummy` module**

`multiprocessing.dummy` replicates the API of `multiprocessing` but is no more than a wrapper around the `threading` module.
16.6.3. Programming guidelines

There are certain guidelines and idioms which should be adhered to when using `multiprocessing`.

16.6.3.1. All platforms

Avoid shared state

As far as possible one should try to avoid shifting large amounts of data between processes.

It is probably best to stick to using queues or pipes for communication between processes rather than using the lower level synchronization primitives from the `threading` module.

Picklability

Ensure that the arguments to the methods of proxies are picklable.

Thread safety of proxies

Do not use a proxy object from more than one thread unless you protect it with a lock.

(There is never a problem with different processes using the same proxy.)

Joining zombie processes

On Unix when a process finishes but has not been joined it becomes a zombie. There should never be very many because each time a new process starts (or `active_children()` is called)
all completed processes which have not yet been joined will be joined. Also calling a finished process’s \texttt{Process.is_alive} will join the process. Even so it is probably good practice to explicitly join all the processes that you start.

Better to inherit than pickle/unpickle

On Windows many types from \texttt{multiprocessing} need to be picklable so that child processes can use them. However, one should generally avoid sending shared objects to other processes using pipes or queues. Instead you should arrange the program so that a process which needs access to a shared resource created elsewhere can inherit it from an ancestor process.

Avoid terminating processes

Using the \texttt{Process.terminate} method to stop a process is liable to cause any shared resources (such as locks, semaphores, pipes and queues) currently being used by the process to become broken or unavailable to other processes.

Therefore it is probably best to only consider using \texttt{Process.terminate} on processes which never use any shared resources.

Joining processes that use queues

Bear in mind that a process that has put items in a queue will wait before terminating until all the buffered items are fed by the “feeder” thread to the underlying pipe. (The child process can call the \texttt{cancel_join_thread()} method of the queue to avoid this behaviour.)

This means that whenever you use a queue you need to make
sure that all items which have been put on the queue will eventually be removed before the process is joined. Otherwise you cannot be sure that processes which have put items on the queue will terminate. Remember also that non-daemonic processes will be joined automatically.

An example which will deadlock is the following:

```python
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()  # this deadlocks
    obj = queue.get()
```

A fix here would be to swap the last two lines (or simply remove the `p.join()` line).

Explicitly pass resources to child processes

On Unix a child process can make use of a shared resource created in a parent process using a global resource. However, it is better to pass the object as an argument to the constructor for the child process.

Apart from making the code (potentially) compatible with Windows this also ensures that as long as the child process is still alive the object will not be garbage collected in the parent process. This might be important if some resource is freed when the object is garbage collected in the parent process.

So for instance
```python
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

should be rewritten as

```python
from multiprocessing import Process, Lock

def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

Beware of replacing `sys.stdin` with a “file like object”

`multiprocessing` originally unconditionally called:

```python
import os
os.close(sys.stdin.fileno())
```

in the `multiprocessing.Process._bootstrap()` method — this resulted in issues with processes-in-processes. This has been changed to:

```python
sys.stdin.close()
sys.stdin = open(os.devnull)
```

Which solves the fundamental issue of processes colliding with each other resulting in a bad file descriptor error, but introduces a potential danger to applications which replace `sys.stdin()` with a “file-like object” with output buffering. This danger is that if
multiple processes call `close()` on this file-like object, it could result in the same data being flushed to the object multiple times, resulting in corruption.

If you write a file-like object and implement your own caching, you can make it fork-safe by storing the pid whenever you append to the cache, and discarding the cache when the pid changes. For example:

```python
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

For more information, see issue 5155, issue 5313 and issue 5331

### 16.6.3.2. Windows

Since Windows lacks `os.fork()` it has a few extra restrictions:

**More picklability**

Ensure that all arguments to `Process.__init__()` are picklable. This means, in particular, that bound or unbound methods cannot be used directly as the `target` argument on Windows — just define a function and use that instead.

Also, if you subclass `Process` then make sure that instances will be picklable when the `Process.start` method is called.

**Global variables**

Bear in mind that if code run in a child process tries to access a
global variable, then the value it sees (if any) may not be the same as the value in the parent process at the time that \texttt{Process.start} was called.

However, global variables which are just module level constants cause no problems.

Safe importing of main module

Make sure that the main module can be safely imported by a new Python interpreter without causing unintended side effects (such a starting a new process).

For example, under Windows running the following module would fail with a \texttt{RuntimeError}:

```python
from multiprocessing import Process

def foo():
    print 'hello'

p = Process(target=foo)
p.start()
```

Instead one should protect the “entry point” of the program by using \texttt{if \_\_name\_ == '__main__':} as follows:

```python
from multiprocessing import Process, freeze_support

def foo():
    print 'hello'

if __name__ == '__main__':
    freeze_support()
    p = Process(target=foo)
    p.start()
```

(The \texttt{freeze_support()} line can be omitted if the program will be run normally instead of frozen.)
This allows the newly spawned Python interpreter to safely import the module and then run the module’s \texttt{foo()} function.

Similar restrictions apply if a pool or manager is created in the main module.
Demonstration of how to create and use customized managers and proxies:

```python
# This module shows how to use arbitrary callables with a subclass
# `BaseManager`.  
# Copyright (c) 2006-2008, R Oudkerk  
# All rights reserved. 
#
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo(object):
    def f(self):
        print 'you called Foo.f()'
    def g(self):
        print 'you called Foo.g()'
    def _h(self):
        print 'you called Foo._h()'

# A simple generator function
def baz():
    for i in xrange(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ('next', '__next__')
    def __iter__(self):
        return self
    def next(self):
        return self._callmethod('next')
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
```
def get_operator_module():
    return operator

class MyManager(BaseManager):  
    pass

# register the Foo class; make `f()` and `g()` accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)

def test():
    manager = MyManager()
    manager.start()

    print '-' * 20

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print '-' * 20

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print '-' * 20

    it = manager.baz()
    for i in it:
        print '<%d>' % i,
Using Pool:

```python
# # A test of `multiprocessing.Pool` class
# # Copyright (c) 2006-2008, R Oudkerk
# # All rights reserved.
# 
import multiprocessing
import time
import random
import sys

# # Functions used by test code
# 
def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
    )

def calculatestar(args):
    return calculate(*args)
```
def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

def f(x):
    return 1.0 / (x-5.0)

def pow3(x):
    return x**3

def noop(x):
    pass

# # Test code ##

def test():
    print 'cpu_count() = %d
' % multiprocessing.cpu_count()

    # # Create pool #

    PROCESSES = 4
    print 'Creating pool with %d processes
' % PROCESSES
    pool = multiprocessing.Pool(PROCESSES)
    print 'pool = %s' % pool
    print

    # # Tests #

    TASKS = [(mul, (i, 7)) for i in range(10)] + 
             [(plus, (i, 8)) for i in range(10)]

    results = [pool.apply_async(calculate, t) for t in TASKS]
    imap_it = pool.imap(calculatestar, TASKS)
    imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

    print 'Ordered results using pool.apply_async():'
    for r in results:
print 'Ordered results using pool.imap():'
for x in imap_it:
    print '	', x
print

print 'Unordered results using pool.imap_unordered():'
for x in imap_unordered_it:
    print '	', x
print

print 'Ordered results using pool.map() --- will block till complete:'
for x in pool.map(calculatestar, TASKS):
    print '	', x
print

# # Simple benchmarks # #

N = 100000
print 'def pow3(x): return x**3'

t = time.time()
A = map(pow3, xrange(N))
print '\tmap(pow3, xrange(%d)):\n\t\t\t\t\t\t\t\t\n\t\t\t\t\t\t\t\t\s seconds' % \n    (N, time.time() - t)

t = time.time()
B = pool.map(pow3, xrange(N))
print '\tpool.map(pow3, xrange(%d)):\n\t\t\t\t\t\t\t\t\n\t\t\t\t\t\t\t\t\s seconds' % \n    (N, time.time() - t)

t = time.time()
C = list(pool.imap(pow3, xrange(N), chunksize=N//8))
print '\tlist(pool.imap(pow3, xrange(%d), chunksize=%d)):\n\t\t\t\t\t\t\t\t\n\t\t\t\t\t\t\t\t\s seconds' % (N, N//8, time.time() - t)

assert A == B == C, (len(A), len(B), len(C))
print

L = [None] * 1000000
print 'def noop(x): pass'
print 'L = [None] * 1000000'
t = time.time()
A = map(noop, L)
print '%s seconds
(time.time() - t)

B = pool.map(noop, L)
print '%s seconds
(time.time() - t)

C = list(pool.imap(noop, L, chunksize=len(L)//8))
print '%s seconds
(len(L)//8, time.time() - t)

assert A == B == C, (len(A), len(B), len(C))
print

del A, B, C, L

# # Test error handling #

print 'Testing error handling:'

try:
    print pool.apply(f, (5,))
except ZeroDivisionError:
    print 'Got ZeroDivisionError as expected from pool.apply()'
else:
    raise AssertionError('expected ZeroDivisionError')

try:
    print pool.map(f, range(10))
except ZeroDivisionError:
    print 'Got ZeroDivisionError as expected from pool.map()'
else:
    raise AssertionError('expected ZeroDivisionError')

try:
    print list(pool.imap(f, range(10)))
except ZeroDivisionError:
    print 'Got ZeroDivisionError as expected from list(pool.imap())'
else:
    raise AssertionError('expected ZeroDivisionError')
it = pool.imap(f, range(10))
for i in range(10):
    try:
        x = it.next()
    except ZeroDivisionError:
        if i == 5:
            pass
        except StopIteration:
            break
    else:
        if i == 5:
            raise AssertionError('expected ZeroDivisionError')
assert i == 9
print '\tGot ZeroDivisionError as expected from IMapIterator'
print

# Testing timeouts
#

print 'Testing ApplyResult.get() with timeout: ',
res = pool.apply_async(calculate, TASKS[0])
while 1:
    sys.stdout.flush()
    try:
        sys.stdout.write('\n\t\t%s' % res.get(0.02))
        break
    except multiprocessing.TimeoutError:
        sys.stdout.write('.')
print
print
print 'Testing IMapIterator.next() with timeout: ',
it = pool.imap(calculateStar, TASKS)
while 1:
    sys.stdout.flush()
    try:
        sys.stdout.write('\n\t\t%s' % it.next(0.02))
    except StopIteration:
        break
    except multiprocessing.TimeoutError:
        sys.stdout.write('.')
print
print
#
# Testing callback
#

print 'Testing callback:'

A = []
B = [56, 0, 1, 8, 27, 64, 125, 216, 343, 512, 729]

r = pool.apply_async(mul, (7, 8), callback=A.append)
r.wait()

r = pool.map_async(pow3, range(10), callback=A.extend)
r.wait()

if A == B:
    print '\tcallbacks succeeded\n'
else:
    print '\t*** callbacks failed\n\t%s != %s' % (A, B)

#
# Check there are no outstanding tasks
#

assert not pool._cache, 'cache = %r' % pool._cache

#
# Check close() methods
#

print 'Testing close():'

for worker in pool._pool:
    assert worker.is_alive()

result = pool.apply_async(time.sleep, [0.5])
pool.close()
pool.join()

assert result.get() is None

for worker in pool._pool:
    assert not worker.is_alive()

print '\tclose() succeeded\n'

#
# Check terminate() method
#

print 'Testing terminate():'

pool = multiprocessing.Pool(2)
DELTA = 0.1
ignore = pool.apply(pow3, [2])
results = [pool.apply_async(time.sleep, [DELTA]) for i in range(pool._pool)]
pool.terminate()
pool.join()

for worker in pool._pool:
    assert not worker.is_alive()

print '\tterminate() succeeded\n'
#
# Check garbage collection
#

print 'Testing garbage collection:'

pool = multiprocessing.Pool(2)
DELTA = 0.1
processes = pool._pool
ignore = pool.apply(pow3, [2])
results = [pool.apply_async(time.sleep, [DELTA]) for i in range(pool._pool)]
results = pool = None
time.sleep(DELTA * 2)

for worker in processes:
    assert not worker.is_alive()

print '\tgarbage collection succeeded\n'

if __name__ == '__main__':
    multiprocessing.freeze_support()

    assert len(sys.argv) in (1, 2)

    if len(sys.argv) == 1 or sys.argv[1] == 'processes':
        print ' Using processes '.center(79, '-')
    elif sys.argv[1] == 'threads':
        print ' Using threads '.center(79, '-')
Synchronization types like locks, conditions and queues:

```python
import multiprocessing

if __name__ == '__main__':
    # A test file for the `multiprocessing` package
    # Copyright (c) 2006-2008, R Oudkerk
    # All rights reserved.

    import time, sys, random
    from Queue import Empty

    import multiprocessing
    # may get overwritten

    #### TEST_VALUE

    def value_func(running, mutex):
        random.seed()
        time.sleep(random.random()*4)

        mutex.acquire()
        print time.strftime('%m%d %t') + str(multiprocessing.current_process()) + 
            running.value -= 1
        mutex.release()

    def test_value():
        TASKS = 10
        running = multiprocessing.Value('i', TASKS)
        mutex = multiprocessing.Lock()

        for i in range(TASKS):
            p = multiprocessing.Process(target=value_func, args=(running,)
                p.start()

        while running.value > 0:
            time.sleep(0.08)
            mutex.acquire()

    test()
```
print running.value,
sys.stdout.flush()
mutex.release()

print
print 'No more running processes'

#### TEST_QUEUE

def queue_func(queue):
    for i in range(30):
        time.sleep(0.5 * random.random())
        queue.put(i*i)
    queue.put('STOP')

def test_queue():
    q = multiprocessing.Queue()

    p = multiprocessing.Process(target=queue_func, args=(q,))
    p.start()

    o = None
    while o != 'STOP':
        try:
            o = q.get(timeout=0.3)
            print o,
            sys.stdout.flush()
        except Empty:
            print 'TIMEOUT'

    print

#### TEST_CONDITION

def condition_func(cond):
    cond.acquire()
    print '	' + str(cond)
    time.sleep(2)
    print '\tchild is notifying'
    print '\t' + str(cond)
    cond.notify()
    cond.release()

def test_condition():
    cond = multiprocessing.Condition()
```python
p = multiprocessing.Process(target=condition_func, args=(cond,))
print cond
cond.acquire()
print cond
cond.acquire()
print cond
p.start()

print 'main is waiting'
cond.wait()
print 'main has woken up'

print cond
cond.release()
print cond
cond.release()

p.join()
print cond

#### TEST_SEMAPHORE

def semaphore_func(sema, mutex, running):
    sema.acquire()

    mutex.acquire()
    running.value += 1
    print running.value, 'tasks are running'
    mutex.release()

    random.seed()
    time.sleep(random.random()*2)

    mutex.acquire()
    running.value -= 1
    print '%s has finished' % multiprocessing.current_process()
    mutex.release()

    sema.release()

def test_semaphore():
    sema = multiprocessing.Semaphore(3)
    mutex = multiprocessing.RLock()
```
running = multiprocessing.Value('i', 0)

processes = [
    multiprocessing.Process(target=semaphore_func,
        args=(sema, mutex, running))
    for i in range(10)
]

for p in processes:
    p.start()

for p in processes:
    p.join()

#### TEST_JOIN_TIMEOUT

def join_timeout_func():
    print 'child sleeping'
    time.sleep(5.5)
    print 'child terminating'

def test_join_timeout():
    p = multiprocessing.Process(target=join_timeout_func)
    p.start()

    print 'waiting for process to finish'

    while 1:
        p.join(timeout=1)
        if not p.is_alive():
            break
        print '.',
        sys.stdout.flush()

#### TEST_EVENT

def event_func(event):
    print '\t%r is waiting' % multiprocessing.current_process()
    event.wait()
    print '\t%r has woken up' % multiprocessing.current_process

def test_event():
    event = multiprocessing.Event()

    processes = [multiprocessing.Process(target=event_func, arg
```python
for i in range(5):
    for p in processes:
        p.start()

print 'main is sleeping'
time.sleep(2)

print 'main is setting event'
event.set()

for p in processes:
    p.join()

### TEST_SHAREDVALUES

def sharedvalues_func(values, arrays, shared_values, shared_arrays):
    for i in range(len(values)):
        v = values[i][1]
        sv = shared_values[i].value
        assert v == sv

    for i in range(len(values)):
        a = arrays[i][1]
        sa = list(shared_arrays[i][:])
        assert a == sa

    print 'Tests passed'

def test_sharedvalues():
    values = [
        ('i', 10),
        ('h', -2),
        ('d', 1.25)
    ]
    arrays = [
        ('i', range(100)),
        ('d', [0.25 * i for i in range(100)]),
        ('H', range(1000))
    ]
    shared_values = [multiprocessing.Value(id, v) for id, v in values]
    shared_arrays = [multiprocessing.Array(id, a) for id, a in arrays]
    p = multiprocessing.Process(
        target=sharedvalues_func,
```
args=(values, arrays, shared_values, shared_arrays)
)
p.start()
p.join()

assert p.exitcode == 0

###

def test(namespace=multiprocessing):
    global multiprocessing

    multiprocessing = namespace

    for func in [test_value, test_queue, test_condition,
                 test_semaphore, test_join_timeout, test_event,
                 test_sharedvalues]:

        print '

    ignore = multiprocessing.active_children()  # cleanup any old processes
    if hasattr(multiprocessing, '_debug_info'):
        info = multiprocessing._debug_info()
        if info:
            print info
            raise ValueError('there should be no positive refcounts left')

    if __name__ == '__main__':
        multiprocessing.freeze_support()

        assert len(sys.argv) in (1, 2)

        if len(sys.argv) == 1 or sys.argv[1] == 'processes':
            print 'Using processes '.center(79, '-')
            namespace = multiprocessing
        elif sys.argv[1] == 'manager':
            print 'Using processes and a manager '.center(79, '-')
            namespace = multiprocessing.Manager()
            namespace.Process = multiprocessing.Process
            namespace.current_process = multiprocessing.current_process
            namespace.active_children = multiprocessing.active_children
        elif sys.argv[1] == 'threads':
            print 'Using threads '.center(79, '-')
            import multiprocessing.dummy as namespace
else:
    print 'Usage:\n%s [processes | manager | threads]' % sys
    raise SystemExit(2)

test(namespace)

An example showing how to use queues to feed tasks to a collection of worker processes and collect the results:

# Simple example which uses a pool of workers to carry out some
# Notice that the results will probably not come out of the output
# queue in the same order as the corresponding tasks were put on the input queue. If it is important to get the results
# in the original order then consider using `Pool.map()` or
# `Pool.imap()` (which will save on the amount of code needed a
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#
import time
import random

from multiprocessing import Process, Queue, current_process, freeze_support

# Function run by worker processes
#
def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

# Function used to calculate result
#
def calculate(func, args):
    result = func(*args)
    return '%%s says that %s = %s' %
            (current_process().name, func.__name__, args, result)
# Functions referenced by tasks
#

def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

# Create queues
task_queue = Queue()
done_queue = Queue()

# Submit tasks
for task in TASKS1:
    task_queue.put(task)

# Start worker processes
for i in range(NUMBER_OF_PROCESSES):
    Process(target=worker, args=(task_queue, done_queue)).start

# Get and print results
print 'Unordered results:'
for i in range(len(TASKS1)):
    print '\t', done_queue.get()

# Add more tasks using `put()`
for task in TASKS2:
    task_queue.put(task)

# Get and print some more results
for i in range(len(TASKS2)):
    print '\t', done_queue.get()

# Tell child processes to stop
for i in range(NUMBER_OF_PROCESSES):
    task_queue.put('STOP')

if __name__ == '__main__':
    freeze_support()
    test()

An example of how a pool of worker processes can each run a
SimpleHTTPServer.HttpServer instance while sharing a single
listening socket.

# Example where a pool of http servers share a single listening
# On Windows this module depends on the ability to pickle a soc
# object so that the worker processes can inherit a copy of the
# object. (We import `multiprocessing.reduction` to enable thi
# Not sure if we should synchronize access to `socket.accept()`
# using a process-shared lock -- does not seem to be necessary.
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#
import os
import sys

from multiprocessing import Process, current_process, freeze_support
from BaseHTTPServer import HTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler

if sys.platform == 'win32':
    import multiprocessing.reduction  # make sockets pickable

def note(format, *args):
    sys.stderr.write('[' + current_process().name + ']

class RequestHandler(SimpleHTTPRequestHandler):
    # we override log_message() to show which process is handling
    def log_message(self, format, *args):
        note(format, *args)
Some simple benchmarks comparing multiprocessing with threading:

```python
# Simple benchmarks for the multiprocessing package
```
import time, sys, multiprocessing, threading, Queue, gc

if sys.platform == 'win32':
    _timer = time.clock
else:
    _timer = time.time

delta = 1

## TEST_QUEUESPEED

def queuespeed_func(q, c, iterations):
    a = '0' * 256
    c.acquire()
    c.notify()
    c.release()

    for i in xrange(iterations):
        q.put(a)

    q.put('STOP')

def test_queuespeed(Process, q, c):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        p = Process(target=queuespeed_func, args=(q, c, iterations)
        c.acquire()
        p.start()
        c.wait()
        c.release()

        result = None
        t = _timer()

        while result != 'STOP':
            result = q.get()
elapsed = _timer() - t
p.join()

print iterations, 'objects passed through connection in', elapsed
print 'average number/sec:', iterations/elapsed

### TEST_PIPESPEED

def pipe_func(c, cond, iterations):
a = '0' * 256
cond.acquire()
cond.notify()
cond.release()

    for i in xrange(iterations):
        c.send(a)

    c.send('STOP')

def test_pipespeed():
c, d = multiprocessing.Pipe()
cond = multiprocessing.Condition()
elapsed = 0
iterations = 1

    while elapsed < delta:
        iterations *= 2

        p = multiprocessing.Process(target=pipe_func,
                                     args=(d, cond, iterations))

        cond.acquire()
p.start()
cond.wait()
cond.release()

        result = None
t = _timer()

        while result != 'STOP':
            result = c.recv()

        elapsed = _timer() - t
p.join()

print iterations, 'objects passed through connection in', elapsed
```python
print 'average number/sec:', iterations/elapsed

#### TEST_SEQSPEED

def test_seqspeed(seq):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        t = _timer()

        for i in xrange(iterations):
            a = seq[5]

        elapsed = _timer()-t

    print iterations, 'iterations in', elapsed, 'seconds'
print 'average number/sec:', iterations/elapsed

#### TEST_LOCK

def test_lockspeed(l):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        t = _timer()

        for i in xrange(iterations):
            l.acquire()
            l.release()

        elapsed = _timer()-t

    print iterations, 'iterations in', elapsed, 'seconds'
print 'average number/sec:', iterations/elapsed

#### TEST_CONDITION

def conditionspeed_func(c, N):
```

c.acquire()
c.notify()

for i in xrange(N):
c.wait()
c.notify()
c.release()

def test_conditionspeed(Process, c):
elapsed = 0
iterations = 1

while elapsed < delta:
    iterations *= 2
    c.acquire()
p = Process(target=conditionspeed_func, args=(c, iterat
p.start()
c.wait()

t = _timer()

for i in xrange(iterations):
c.notify()
c.wait()

    elapsed = _timer()-t
c.release()
p.join()

print iterations * 2, 'waits in', elapsed, 'seconds'
print 'average number/sec:', iterations * 2 / elapsed

####

def test():
    manager = multiprocessing.Manager()

gc.disable()

print '\n\t##### testing Queue.Queue\n'
test_queuespeed(threading.Thread, Queue.Queue(),
    threading.Condition())
print '\n\t##### testing multiprocessing.Queue\n'
test_queuespeed(multiprocessing.Process, multiprocessing.Queue(), multiprocessing.Condition())
print '\n\t######## testing Queue managed by server process\
test_queuespeed(multiprocessing.Process, manager.Queue(), manager.Condition())
print '\n\t######## testing multiprocessing.Pipe

test_pipespeed()

print '\n\t######## testing list

test_seqspeed(range(10))
print '\n\t######## testing list managed by server process\
test_seqspeed(manager.list(range(10)))
print '\n\t######## testing Array("i", ..., lock=False)\n'test_seqspeed(multiprocessing.Array('i', range(10), lock=False))
print '\n\t######## testing Array("i", ..., lock=True)\n'test_seqspeed(multiprocessing.Array('i', range(10), lock=True))

print '\n\t######## testing threading.Lock
'test_lockspeed(threading.Lock())
print '\n\t######## testing threading.RLock
'test_lockspeed(threading.RLock())
print '\n\t######## testing multiprocessing.Lock
'test_lockspeed(multiprocessing.Lock())
print '\n\t######## testing multiprocessing.RLock
'test_lockspeed(multiprocessing.RLock())
print '\n\t######## testing lock managed by server process\
test_lockspeed(manager.Lock())
print '\n\t######## testing rlock managed by server process
'test_lockspeed(manager.RLock())

print '\n\t######## testing threading.Condition
'test_conditionspeed(threading.Thread, threading.Condition())
print '\n\t######## testing multiprocessing.Condition\n'test_conditionspeed(multiprocessing.Process, multiprocessing.Condition())
print '\n\t######## testing condition managed by a server process\n'test_conditionspeed(multiprocessing.Process, manager.Condition())

if __name__ == '__main__':
multiprocessing.freeze_support()
The Python Standard Library » 16. Optional Operating System Services »
16.7. `mmap` — Memory-mapped file support

Memory-mapped file objects behave like both strings and like file objects. Unlike normal string objects, however, these are mutable. You can use `mmap` objects in most places where strings are expected; for example, you can use the `re` module to search through a memory-mapped file. Since they're mutable, you can change a single character by doing `obj[index] = 'a'`, or change a substring by assigning to a slice: `obj[i1:i2] = '...'. You can also read and write data starting at the current file position, and `seek()` through the file to different positions.

A memory-mapped file is created by the `mmap` constructor, which is different on Unix and on Windows. In either case you must provide a file descriptor for a file opened for update. If you wish to map an existing Python file object, use its `fileno()` method to obtain the correct value for the `fileno` parameter. Otherwise, you can open the file using the `os.open()` function, which returns a file descriptor directly (the file still needs to be closed when done).

**Note:** If you want to create a memory-mapping for a writable, buffered file, you should `flush()` the file first. This is necessary to ensure that local modifications to the buffers are actually available to the mapping.

For both the Unix and Windows versions of the constructor, `access` may be specified as an optional keyword parameter. `access` accepts one of three values: `ACCESS_READ`, `ACCESS_WRITE`, or `ACCESS_COPY` to specify read-only, write-through or copy-on-write memory respectively. `access` can be used on both Unix and Windows. If
access is not specified, Windows mmap returns a write-through mapping. The initial memory values for all three access types are taken from the specified file. Assignment to an ACCESS_READ memory map raises a TypeError exception. Assignment to an ACCESS_WRITE memory map affects both memory and the underlying file. Assignment to an ACCESS_COPY memory map affects memory but does not update the underlying file.

Changed in version 2.5: To map anonymous memory, -1 should be passed as the fileno along with the length.

Changed in version 2.6: mmap.mmap has formerly been a factory function creating mmap objects. Now mmap.mmap is the class itself.

class mmap.mmap(fileno, length[, tagname[, access[, offset]]]])

(Windows version) Maps length bytes from the file specified by the file handle fileno, and creates a mmap object. If length is larger than the current size of the file, the file is extended to contain length bytes. If length is 0, the maximum length of the map is the current size of the file, except that if the file is empty Windows raises an exception (you cannot create an empty mapping on Windows).

tagname, if specified and not None, is a string giving a tag name for the mapping. Windows allows you to have many different mappings against the same file. If you specify the name of an existing tag, that tag is opened, otherwise a new tag of this name is created. If this parameter is omitted or None, the mapping is created without a name. Avoiding the use of the tag parameter will assist in keeping your code portable between Unix and Windows.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the
file. *offset* defaults to 0. *offset* must be a multiple of the ALLOCATIONGRANULARITY.

```python
class mmap(mmap(fileno, length[, flags[, prot[, access[, offset]]]]))
```

*(Unix version)* Maps *length* bytes from the file specified by the file descriptor *fileno*, and returns a mmap object. If *length* is 0, the maximum length of the map will be the current size of the file when *mmap* is called.

*flags* specifies the nature of the mapping. MAP_PRIVATE creates a private copy-on-write mapping, so changes to the contents of the mmap object will be private to this process, and MAP_SHARED creates a mapping that’s shared with all other processes mapping the same areas of the file. The default value is MAP_SHARED.

*prot*, if specified, gives the desired memory protection; the two most useful values are PROT_READ and PROT_WRITE, to specify that the pages may be read or written. *prot* defaults to PROT_READ | PROT_WRITE.

*access* may be specified in lieu of *flags* and *prot* as an optional keyword parameter. It is an error to specify both *flags*, *prot* and *access*. See the description of *access* above for information on how to use this parameter.

*offset* may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of the PAGESIZE or ALLOCATIONGRANULARITY.

To ensure validity of the created memory mapping the file specified by the descriptor *fileno* is internally automatically synchronized with physical backing store on Mac OS X and
This example shows a simple way of using `mmap`:

```python
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write("Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print mm.readline()  # prints "Hello Python!"
    # read content via slice notation
    print mm[:5]  # prints "Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = " world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print mm.readline()  # prints "Hello  world!"
    # close the map
    mm.close()
```

The next example demonstrates how to create an anonymous map and exchange data between the parent and child processes:

```python
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write("Hello world!"

pid = os.fork()

if pid == 0:  # In a child process
    mm.seek(0)
    print mm.readline()
    mm.close()
```

Memory-mapped file objects support the following methods:
close()
Closes the mmap. Subsequent calls to other methods of the
object will result in a ValueError exception being raised. This
will not close the open file.

find(string[, start[, end]])
Returns the lowest index in the object where the substring
string is found, such that string is contained in the range [start,
end]. Optional arguments start and end are interpreted as in
slice notation. Returns -1 on failure.

flush([offset, size])
Flushes changes made to the in-memory copy of a file back to
disk. Without use of this call there is no guarantee that
changes are written back before the object is destroyed. If
offset and size are specified, only changes to the given range
of bytes will be flushed to disk; otherwise, the whole extent of
the mapping is flushed.

(Windows version) A nonzero value returned indicates
success; zero indicates failure.

(Unix version) A zero value is returned to indicate success.
An exception is raised when the call failed.

move(dest, src, count)
Copy the count bytes starting at offset src to the destination
index dest. If the mmap was created with ACCESS_READ, then
calls to move will raise a TypeError exception.

read(num)
Return a string containing up to num bytes starting from the
current file position; the file position is updated to point after
the bytes that were returned.
read_byte()
Returns a string of length 1 containing the character at the current file position, and advances the file position by 1.

readline()
Returns a single line, starting at the current file position and up to the next newline.

resize(newsize)
Resizes the map and the underlying file, if any. If the mmap was created with ACCESS_READ or ACCESS_COPY, resizing the map will raise a TypeError exception.

rfind(string[, start[, end]])
Returns the highest index in the object where the substring string is found, such that string is contained in the range [start, end]. Optional arguments start and end are interpreted as in slice notation. Returns -1 on failure.

seek(pos[, whence])
Set the file's current position. whence argument is optional and defaults to os.SEEK_SET or 0 (absolute file positioning); other values are os.SEEK_CUR or 1 (seek relative to the current position) and os.SEEK_END or 2 (seek relative to the file's end).

size()
Return the length of the file, which can be larger than the size of the memory-mapped area.

tell()
Returns the current position of the file pointer.

write(string)
Write the bytes in string into memory at the current position of
the file pointer; the file position is updated to point after the bytes that were written. If the mmap was created with 
ACCESS_READ, then writing to it will raise a TypeError exception.

```
write_byte(byte)
```

Write the single-character string `byte` into memory at the current position of the file pointer; the file position is advanced by 1. If the mmap was created with ACCESS_READ, then writing to it will raise a TypeError exception.
The `readline` module defines a number of functions to facilitate completion and reading/writing of history files from the Python interpreter. This module can be used directly, or via the `rlcompleter` module, which supports completion of Python identifiers at the interactive prompt. Settings made using this module affect the behaviour of both the interpreter's interactive prompt and the prompts offered by the `raw_input()` and `input()` built-in functions.

**Note:** The underlying Readline library API may be implemented by the `libedit` library instead of GNU readline. On MacOS X the `readline` module detects which library is being used at run time.

The configuration file for `libedit` is different from that of GNU readline. If you programmatically load configuration strings you can check for the text “libedit” in `readline.__doc__` to differentiate between GNU readline and libedit.

Readline keybindings may be configured via an initialization file, typically `.inputrc` in your home directory. See Readline Init File in the GNU Readline manual for information about the format and allowable constructs of that file, and the capabilities of the Readline library in general.
16.8.1. Init file

The following functions relate to the init file and user configuration:

`readline.parse_and_bind(string)`
Execute the init line provided in the `string` argument. This calls `rl_parse_and_bind()` in the underlying library.

`readline.read_init_file([filename])`
Execute a readline initialization file. The default filename is the last filename used. This calls `rl_read_init_file()` in the underlying library.
16.8.2. Line buffer

The following functions operate on the line buffer:

readline. **get_line_buffer()**
Return the current contents of the line buffer (`rl_line_buffer` in the underlying library).

readline. **insert_text(string)**
Insert text into the line buffer at the cursor position. This calls `rl_insert_text()` in the underlying library, but ignores the return value.

readline. **redisplay()**
Change what's displayed on the screen to reflect the current contents of the line buffer. This calls `rl_redisplay()` in the underlying library.
16.8.3. History file

The following functions operate on a history file:

**readline.** read_history_file([filename])

Load a readline history file, and append it to the history list. The default filename is ~/.history. This calls read_history() in the underlying library.

**readline.** write_history_file([filename])

Save the history list to a readline history file, overwriting any existing file. The default filename is ~/.history. This calls write_history() in the underlying library.

**readline.** get_history_length()
**readline.** set_history_length(length)

Set or return the desired number of lines to save in the history file. The write_history_file() function uses this value to truncate the history file, by calling history_truncate_file() in the underlying library. Negative values imply unlimited history file size.
16.8.4. History list

The following functions operate on a global history list:

**readline.**\texttt{clear\_history()}  
Clear the current history. This calls \texttt{clear\_history()} in the underlying library. The Python function only exists if Python was compiled for a version of the library that supports it.

*New in version 2.4.*

**readline.**\texttt{get\_current\_history\_length()}  
Return the number of items currently in the history. (This is different from \texttt{get\_history\_length()}, which returns the maximum number of lines that will be written to a history file.)

*New in version 2.3.*

**readline.**\texttt{get\_history\_item(index)}  
Return the current contents of history item at \texttt{index}. The item index is one-based. This calls \texttt{history\_get()} in the underlying library.

*New in version 2.3.*

**readline.**\texttt{remove\_history\_item(pos)}  
Remove history item specified by its position from the history. The position is zero-based. This calls \texttt{remove\_history()} in the underlying library.

*New in version 2.4.*

**readline.**\texttt{replace\_history\_item(pos, line)}  
Replace history item specified by its position with \texttt{line}. The
position is zero-based. This calls replace_history_entry() in the underlying library.

*New in version 2.4.*

`readline.add_history(line)`

Append `line` to the history buffer, as if it was the last line typed. This calls `add_history()` in the underlying library.
16.8.5. Startup hooks

*New in version 2.3.*

**readline.set_startup_hook**([function])

Set or remove the function invoked by the `rl_startup_hook` callback of the underlying library. If `function` is specified, it will be used as the new hook function; if omitted or `None`, any function already installed is removed. The hook is called with no arguments just before `readline` prints the first prompt.

**readline.set_pre_input_hook**([function])

Set or remove the function invoked by the `rl_pre_input_hook` callback of the underlying library. If `function` is specified, it will be used as the new hook function; if omitted or `None`, any function already installed is removed. The hook is called with no arguments after the first prompt has been printed and just before `readline` starts reading input characters. This function only exists if Python was compiled for a version of the library that supports it.
The following functions relate to implementing a custom word completion function. This is typically operated by the Tab key, and can suggest and automatically complete a word being typed. By default, Readline is set up to be used by `rlcompleter` to complete Python identifiers for the interactive interpreter. If the `readline` module is to be used with a custom completer, a different set of word delimiters should be set.

```python
readline.set_completer([function])
```

Set or remove the completer function. If `function` is specified, it will be used as the new completer function; if omitted or `None`, any completer function already installed is removed. The completer function is called as `function(text, state)`, for `state` in 0, 1, 2, ..., until it returns a non-string value. It should return the next possible completion starting with `text`.

The installed completer function is invoked by the `entry_func` callback passed to `rl_completion_matches()` in the underlying library. The `text` string comes from the first parameter to the `rl_attempted_completion_function` callback of the underlying library.

```python
readline.get_completer()
```

Get the completer function, or `None` if no completer function has been set.

*New in version 2.3.*

```python
readline.get_completion_type()
```

Get the type of completion being attempted. This returns the
**rl_completion_type** variable in the underlying library as an integer.

*New in version 2.6.*

```python
readline.get_begidx()
readline.get_endidx()
```

Get the beginning or ending index of the completion scope. These indexes are the *start* and *end* arguments passed to the `rl_attempted_completion_function` callback of the underlying library.

```python
readline.set_completer_delims(string)
readline.get_completer_delims()
```

Set or get the word delimiters for completion. These determine the start of the word to be considered for completion (the completion scope). These functions access the `rl_completer_word_break_characters` variable in the underlying library.

```python
readline.set_completion_display_matches_hook([function])
```

Set or remove the completion display function. If *function* is specified, it will be used as the new completion display function; if omitted or `None`, any completion display function already installed is removed. This sets or clears the `rl_completion_display_matches_hook` callback in the underlying library. The completion display function is called as `function(substitution, [matches], longest_match_length)` once each time matches need to be displayed.

*New in version 2.6.*
The following example demonstrates how to use the `readline` module’s history reading and writing functions to automatically load and save a history file named `.pyhist` from the user’s home directory. The code below would normally be executed automatically during interactive sessions from the user’s `PYTHONSTARTUP` file.

```python
import os
import readline
histfile = os.path.join(os.path.expanduser("~"), ".pyhist")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow unruly
    readline.set_history_length(1000)
except IOError:
    pass
import atexit
atexit.register(readline.write_history_file, histfile)
del os, histfile
```

The following example extends the `code.InteractiveConsole` class to support history save/restore.

```python
import code
import readline
import atexit
import os

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self, locals, filename,
                                          histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
```
```python
def save_history(self, histfile):
    readline.set_history_length(1000)
    readline.write_history_file(histfile)
```

```python
except IOError:
    pass
atexit.register(self.save_history, histfile)
```
The `rlcompleter` module defines a completion function suitable for the `readline` module by completing valid Python identifiers and keywords.

When this module is imported on a Unix platform with the `readline` module available, an instance of the `Completer` class is automatically created and its `complete()` method is set as the `readline` completer.

Example:

```python
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline.<TAB PRESSED>
readline.__doc__    readline.get_line_buffer(
readline.__file__   readline.insert_text(
readline.__name__   readline.parse_and_bind(
>>> readline.
```

The `rlcompleter` module is designed for use with Python's interactive mode. A user can add the following lines to his or her initialization file (identified by the `PYTHONSTARTUP` environment variable) to get automatic Tab completion:

```
try:
    import readline
except ImportError:
    print "Module readline not available."
else:
```
On platforms without `readline`, the `Completer` class defined by this module can still be used for custom purposes.
16.9.1. Completer Objects

Completer objects have the following method:

Completer.complete(text, state)

Return the state\textsuperscript{th} completion for text.

If called for text that doesn't include a period character (',.'), it will complete from names currently defined in __main__, __builtin__ and keywords (as defined by the keyword module).

If called for a dotted name, it will try to evaluate anything without obvious side-effects (functions will not be evaluated, but it can generate calls to __getattr__()) up to the last part, and find matches for the rest via the dir() function. Any exception raised during the evaluation of the expression is caught, silenced and None is returned.
17. Interprocess Communication and Networking

The modules described in this chapter provide mechanisms for different processes to communicate.

Some modules only work for two processes that are on the same machine, e.g. `signal` and `subprocess`. Other modules support networking protocols that two or more processes can use to communicate across machines.

The list of modules described in this chapter is:

- **17.1. subprocess — Subprocess management**
  - 17.1.1. Using the `subprocess` Module
    - 17.1.1.1. Frequently Used Arguments
    - 17.1.1.2. Popen Constructor
    - 17.1.1.3. Exceptions
    - 17.1.1.4. Security
  - 17.1.2. Popen Objects
  - 17.1.3. Windows Popen Helpers
    - 17.1.3.1. Constants
  - 17.1.4. Replacing Older Functions with the `subprocess` Module
    - 17.1.4.1. Replacing `/bin/sh` shell backquote
    - 17.1.4.2. Replacing shell pipeline
    - 17.1.4.3. Replacing `os.system()`
    - 17.1.4.4. Replacing the `os.spawn` family
    - 17.1.4.5. Replacing `os.popen()`, `os.popen2()`, `os.popen3()`
    - 17.1.4.6. Replacing functions from the `popen2` module
  - 17.1.5. Notes
- 17.1.5.1. Converting an argument sequence to a string on Windows
- 17.2. socket — Low-level networking interface
  - 17.2.1. Socket Objects
  - 17.2.2. Example
- 17.3. ss1 — TLS/SSL wrapper for socket objects
  - 17.3.1. Functions, Constants, and Exceptions
    - 17.3.1.1. Socket creation
    - 17.3.1.2. Context creation
    - 17.3.1.3. Random generation
    - 17.3.1.4. Certificate handling
    - 17.3.1.5. Constants
  - 17.3.2. SSL Sockets
  - 17.3.3. SSL Contexts
  - 17.3.4. Certificates
    - 17.3.4.1. Certificate chains
    - 17.3.4.2. CA certificates
    - 17.3.4.3. Combined key and certificate
    - 17.3.4.4. Self-signed certificates
  - 17.3.5. Examples
    - 17.3.5.1. Testing for SSL support
    - 17.3.5.2. Client-side operation
    - 17.3.5.3. Server-side operation
  - 17.3.6. Notes on non-blocking sockets
  - 17.3.7. Security considerations
    - 17.3.7.1. Best defaults
    - 17.3.7.2. Manual settings
      - 17.3.7.2.1. Verifying certificates
      - 17.3.7.2.2. Protocol versions
      - 17.3.7.2.3. Cipher selection
    - 17.3.7.3. Multi-processing
- 17.4. signal — Set handlers for asynchronous events
  - 17.4.1. Example
- 17.5. popen2 — Subprocesses with accessible I/O streams
• 17.5.1. Popen3 and Popen4 Objects
• 17.5.2. Flow Control Issues
• 17.6. asyncore — Asynchronous socket handler
  ○ 17.6.1. asyncore Example basic HTTP client
  ○ 17.6.2. asyncore Example basic echo server
• 17.7. asynchat — Asynchronous socket command/response handler
  ○ 17.7.1. asynchat - Auxiliary Classes
  ○ 17.7.2. asynchat Example
17.1. subprocess — Subprocess management

*New in version 2.4.*

The `subprocess` module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module intends to replace several older modules and functions:

```python
os.system
os.spawn*
os.popen*
popen2.*
commands.*
```

Information about how this module can be used to replace the older functions can be found in the [subprocess-replacements] section.

**See also:** POSIX users (Linux, BSD, etc.) are strongly encouraged to install and use the much more recent `subprocess32` module instead of the version included with Python 2.7. It is a drop in replacement with better behavior in many situations.

[PEP 324] – PEP proposing the subprocess module
17.1.1. Using the subprocess Module

The recommended way to launch subprocesses is to use the following convenience functions. For more advanced use cases when these do not meet your needs, use the underlying `Popen` interface.

`subprocess.call(args, *, stdin=None, stdout=None, stderr=None, shell=False)`

Run the command described by `args`. Wait for command to complete, then return the `returncode` attribute.

The arguments shown above are merely the most common ones, described below in Frequently Used Arguments (hence the slightly odd notation in the abbreviated signature). The full function signature is the same as that of the `Popen` constructor - this functions passes all supplied arguments directly through to that interface.

Examples:

```python
>>> subprocess.call(['ls', '-l'])
0
```

```python
>>> subprocess.call('exit 1', shell=True)
1
```

**Warning:** Using `shell=True` can be a security hazard. See the warning under Frequently Used Arguments for details.

**Note:** Do not use `stdout=PIPE` or `stderr=PIPE` with this function as that can deadlock based on the child process output volume. Use `Popen` with the `communicate()` method when you need pipes.
Run command with arguments. Wait for command to complete. If the return code was zero then return, otherwise raise `CalledProcessError`. The `CalledProcessError` object will have the return code in the `returncode` attribute.

The arguments shown above are merely the most common ones, described below in Frequently Used Arguments (hence the slightly odd notation in the abbreviated signature). The full function signature is the same as that of the `Popen` constructor - this functions passes all supplied arguments directly through to that interface.

Examples:

```python
g>>> subprocess.check_call(['ls', '-l'])
0

g>>> subprocess.check_call("exit 1", shell=True)
Traceback (most recent call last):
  ... subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1
```

New in version 2.5.

**Warning:** Using `shell=True` can be a security hazard. See the warning under Frequently Used Arguments for details.

**Note:** Do not use `stdout=PIPE` or `stderr=PIPE` with this function as that can deadlock based on the child process output volume. Use `Popen` with the `communicate()` method when you need pipes.
shell=False, universal_newlines=False)

Run command with arguments and return its output as a byte string.

If the return code was non-zero it raises a CalledProcessError. The CalledProcessError object will have the return code in the returncode attribute and any output in the output attribute.

The arguments shown above are merely the most common ones, described below in Frequently Used Arguments (hence the slightly odd notation in the abbreviated signature). The full function signature is largely the same as that of the Popen constructor, except that stdout is not permitted as it is used internally. All other supplied arguments are passed directly through to the Popen constructor.

Examples:

```python
>>> subprocess.check_output(['echo', 'Hello World!'])
'Hello World!\n'

>>> subprocess.check_output(['exit 1'], shell=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

To also capture standard error in the result, use stderr=subprocess.STDOUT:

```python
>>> subprocess.check_output(...
  ...
  'ls non_existent_file; exit 0',
  ...
  stderr=subprocess.STDOUT,
  ...
  shell=True)
'ls: non_existent_file: No such file or directory\n'
```

New in version 2.7.
**Warning:** Using `shell=True` can be a security hazard. See the warning under *Frequently Used Arguments* for details.

**Note:** Do not use `stderr=PIPE` with this function as that can deadlock based on the child process error volume. Use `Popen` with the `communicate()` method when you need a stderr pipe.

`subprocess.PIPE`

Special value that can be used as the `stdin`, `stdout` or `stderr` argument to `Popen` and indicates that a pipe to the standard stream should be opened.

`subprocess.STDOUT`

Special value that can be used as the `stderr` argument to `Popen` and indicates that standard error should go into the same handle as standard output.

`exception subprocess.CalledProcessError`

Exception raised when a process run by `check_call()` or `check_output()` returns a non-zero exit status.

`returncode`

Exit status of the child process.

`cmd`

Command that was used to spawn the child process.

`output`

Output of the child process if this exception is raised by `check_output()`. Otherwise, `None`.

### 17.1.1.1. Frequently Used Arguments

To support a wide variety of use cases, the `Popen` constructor (and
the convenience functions) accept a large number of optional arguments. For most typical use cases, many of these arguments can be safely left at their default values. The arguments that are most commonly needed are:

- **args** is required for all calls and should be a string, or a sequence of program arguments. Providing a sequence of arguments is generally preferred, as it allows the module to take care of any required escaping and quoting of arguments (e.g. to permit spaces in file names). If passing a single string, either **shell** must be True (see below) or else the string must simply name the program to be executed without specifying any arguments.

- **stdin**, **stdout** and **stderr** specify the executed program’s standard input, standard output and standard error file handles, respectively. Valid values are PIPE, an existing file descriptor (a positive integer), an existing file object, and None. PIPE indicates that a new pipe to the child should be created. With the default settings of None, no redirection will occur; the child’s file handles will be inherited from the parent. Additionally, **stderr** can be STDOUT, which indicates that the stderr data from the child process should be captured into the same file handle as for stdout.

When **stdout** or **stderr** are pipes and **universal_newlines** is True then all line endings will be converted to \n as described for the universal newlines 'u' mode argument to open().

If **shell** is True, the specified command will be executed through the shell. This can be useful if you are using Python primarily for the enhanced control flow it offers over most system shells and still want convenient access to other shell features such as shell pipes, filename wildcards, environment variable expansion, and
expansion of ~ to a user’s home directory. However, note that Python itself offers implementations of many shell-like features (in particular, `glob`, `fnmatch`, `os.walk()`, `os.path.expandvars()`, `os.path.expanduser()`, and `shutil`).

**Warning:** Executing shell commands that incorporate unsanitized input from an untrusted source makes a program vulnerable to shell injection, a serious security flaw which can result in arbitrary command execution. For this reason, the use of `shell=True` is **strongly discouraged** in cases where the command string is constructed from external input:

```python
>>> from subprocess import call
>>> filename = input("What file would you like to display?"
What file would you like to display?
non_existent; rm -rf / #
>>> call("cat " + filename, shell=True) # Uh-oh. This will
```

`shell=False` disables all shell based features, but does not suffer from this vulnerability; see the Note in the `Popen` constructor documentation for helpful hints in getting `shell=False` to work.

When using `shell=True`, `pipes.quote()` can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

These options, along with all of the other options, are described in more detail in the `Popen` constructor documentation.

### 17.1.1.2. Popen Constructor

The underlying process creation and management in this module is handled by the `Popen` class. It offers a lot of flexibility so that
developers are able to handle the less common cases not covered by the convenience functions.

```python
class subprocess.Popen(args, bufsize=0, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=False, shell=False, cwd=None, env=None, universal_newlines=False, startupinfo=None, creationflags=0)
```

Execute a child program in a new process. On Unix, the class uses `os.execvp()`-like behavior to execute the child program. On Windows, the class uses the Windows `CreateProcess()` function. The arguments to `Popen` are as follows.

`args` should be a sequence of program arguments or else a single string. By default, the program to execute is the first item in `args` if `args` is a sequence. If `args` is a string, the interpretation is platform-dependent and described below. See the `shell` and `executable` arguments for additional differences from the default behavior. Unless otherwise stated, it is recommended to pass `args` as a sequence.

On Unix, if `args` is a string, the string is interpreted as the name or path of the program to execute. However, this can only be done if not passing arguments to the program.

**Note:** `shlex.split()` can be useful when determining the correct tokenization for `args`, especially in complex cases:

```python
>>> import shlex, subprocess
>>> command_line = raw_input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd '
>>> args = shlex.split(command_line)
>>> print args
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd']
>>> p = subprocess.Popen(args) # Success!
```

Note in particular that options (such as `-input`) and arguments (such as `eggs.txt`) that are separated by whitespace in the shell
go in separate list elements, while arguments that need quoting or backslash escaping when used in the shell (such as filenames containing spaces or the echo command shown above) are single list elements.

On Windows, if args is a sequence, it will be converted to a string in a manner described in Converting an argument sequence to a string on Windows. This is because the underlying CreateProcess() operates on strings.

The shell argument (which defaults to False) specifies whether to use the shell as the program to execute. If shell is True, it is recommended to pass args as a string rather than as a sequence.

On Unix with shell=True, the shell defaults to /bin/sh. If args is a string, the string specifies the command to execute through the shell. This means that the string must be formatted exactly as it would be when typed at the shell prompt. This includes, for example, quoting or backslash escaping filenames with spaces in them. If args is a sequence, the first item specifies the command string, and any additional items will be treated as additional arguments to the shell itself. That is to say, Popen does the equivalent of:

```python
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

On Windows with shell=True, the COMSPEC environment variable specifies the default shell. The only time you need to specify shell=True on Windows is when the command you wish to execute is built into the shell (e.g. dir or copy). You do not need shell=True to run a batch file or console-based executable.

**Warning:** Passing shell=True can be a security hazard if
combined with untrusted input. See the warning under Frequently Used Arguments for details.

`bufsize`, if given, has the same meaning as the corresponding argument to the built-in open() function: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size. A negative `bufsize` means to use the system default, which usually means fully buffered. The default value for `bufsize` is 0 (unbuffered).

**Note:** If you experience performance issues, it is recommended that you try to enable buffering by setting `bufsize` to either -1 or a large enough positive value (such as 4096).

The `executable` argument specifies a replacement program to execute. It is very seldom needed. When `shell=False`, `executable` replaces the program to execute specified by `args`. However, the original `args` is still passed to the program. Most programs treat the program specified by `args` as the command name, which can then be different from the program actually executed. On Unix, the `args` name becomes the display name for the executable in utilities such as `ps`. If `shell=True`, on Unix the `executable` argument specifies a replacement shell for the default `/bin/sh`.

`stdin`, `stdout` and `stderr` specify the executed program’s standard input, standard output and standard error file handles, respectively. Valid values are `PIPE`, an existing file descriptor (a positive integer), an existing file object, and `None`. `PIPE` indicates that a new pipe to the child should be created. With the default settings of `None`, no redirection will occur; the child’s file handles will be inherited from the parent. Additionally, `stderr` can be `STDOUT`, which indicates that the stderr data from the child process should be captured into the same file handle as for
If `preexec_fn` is set to a callable object, this object will be called in the child process just before the child is executed. (Unix only)

If `close_fds` is true, all file descriptors except 0, 1 and 2 will be closed before the child process is executed. (Unix only). Or, on Windows, if `close_fds` is true then no handles will be inherited by the child process. Note that on Windows, you cannot set `close_fds` to true and also redirect the standard handles by setting `stdin`, `stdout` or `stderr`.

If `cwd` is not `None`, the child’s current directory will be changed to `cwd` before it is executed. Note that this directory is not considered when searching the executable, so you can’t specify the program’s path relative to `cwd`.

If `env` is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of inheriting the current process’ environment, which is the default behavior.

**Note:** If specified, `env` must provide any variables required for the program to execute. On Windows, in order to run a side-by-side assembly the specified `env` must include a valid `SystemRoot`.

If `universal_newlines` is `True`, the file objects `stdout` and `stderr` are opened as text files in `universal newlines` mode. Lines may be terminated by any of `\n`, the Unix end-of-line convention, `\r`, the old Macintosh convention or `\r\n`, the Windows convention. All of these external representations are seen as `\n` by the Python program.
Note: This feature is only available if Python is built with universal newline support (the default). Also, the newlines attribute of the file objects `stdout`, `stdin` and `stderr` are not updated by the `communicate()` method.

If given, `startupinfo` will be a `STARTUPINFO` object, which is passed to the underlying `CreateProcess` function. `creationflags`, if given, can be `CREATE_NEW_CONSOLE` or `CREATE_NEW_PROCESS_GROUP`. (Windows only)

### 17.1.1.3. Exceptions

Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent. Additionally, the exception object will have one extra attribute called `child_traceback`, which is a string containing traceback information from the child's point of view.

The most common exception raised is `OSError`. This occurs, for example, when trying to execute a non-existent file. Applications should prepare for `OSError` exceptions.

A `ValueError` will be raised if `Popen` is called with invalid arguments.

`check_call()` and `check_output()` will raise `CalledProcessError` if the called process returns a non-zero return code.

### 17.1.1.4. Security

Unlike some other `popen` functions, this implementation will never call a system shell implicitly. This means that all characters, including shell metacharacters, can safely be passed to child processes. Obviously, if the shell is invoked explicitly, then it is the application's responsibility to ensure that all whitespace and metacharacters are
quoted appropriately.
17.1.2. Popen Objects

Instances of the `Popen` class have the following methods:

Popen. `poll()`
Check if child process has terminated. Set and return `returncode` attribute.

Popen. `wait()`
Wait for child process to terminate. Set and return `returncode` attribute.

**Warning:** This will deadlock when using `stdout=PIPE` and/or `stderr=PIPE` and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use `communicate()` to avoid that.

Popen. `communicate(input=None)`
Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate. The optional `input` argument should be a string to be sent to the child process, or `None`, if no data should be sent to the child.

`communicate()` returns a tuple `(stdoutdata, stderrdata)`.

Note that if you want to send data to the process's stdin, you need to create the Popen object with `stdin=PIPE`. Similarly, to get anything other than `None` in the result tuple, you need to give `stdout=PIPE` and/or `stderr=PIPE` too.

**Note:** The data read is buffered in memory, so do not use this
method if the data size is large or unlimited.

Popen. **send_signal**(signal)

Sends the signal signal to the child.

**Note:** On Windows, SIGTERM is an alias for terminate().
CTRL_C_EVENT and CTRL_BREAK_EVENT can be sent to processes started with a creationflags parameter which includes CREATE_NEW_PROCESS_GROUP.

*New in version 2.6.*

Popen. **terminate()**

Stop the child. On Posix OSs the method sends SIGTERM to the child. On Windows the Win32 API function TerminateProcess() is called to stop the child.

*New in version 2.6.*

Popen. **kill()**

Kills the child. On Posix OSs the function sends SIGKILL to the child. On Windows kill() is an alias for terminate().

*New in version 2.6.*

The following attributes are also available:

**Warning:** Use communicate() rather than .stdin.write, .stdout.read or .stderr.read to avoid deadlocks due to any of the other OS pipe buffers filling up and blocking the child process.

Popen. **stdin**

If the stdin argument was PIPE, this attribute is a file object that provides input to the child process. Otherwise, it is None.
Popen. **stdout**

If the `stdout` argument was `PIPE`, this attribute is a file object that provides output from the child process. Otherwise, it is `None`.

Popen. **stderr**

If the `stderr` argument was `PIPE`, this attribute is a file object that provides error output from the child process. Otherwise, it is `None`.

Popen. **pid**

The process ID of the child process.

Note that if you set the `shell` argument to `True`, this is the process ID of the spawned shell.

Popen. **returncode**

The child return code, set by `poll()` and `wait()` (and indirectly by `communicate()`). A `None` value indicates that the process hasn’t terminated yet.

A negative value `-N` indicates that the child was terminated by signal `N` (Unix only).
17.1.3. Windows Popen Helpers

The `STARTUPINFO` class and following constants are only available on Windows.

```python
class subprocess.STARTUPINFO
    Partial support of the Windows STARTUPINFO structure is used for Popen creation.

    dwFlags
        A bit field that determines whether certain STARTUPINFO attributes are used when the process creates a window.

        `si = subprocess.STARTUPINFO()`
        `si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_USESHOWWINDOW`
```

hStdInput
    If `dwFlags` specifies `STARTF_USESTDHANDLES`, this attribute is the standard input handle for the process. If `STARTF_USESTDHANDLES` is not specified, the default for standard input is the keyboard buffer.

hStdOutput
    If `dwFlags` specifies `STARTF_USESTDHANDLES`, this attribute is the standard output handle for the process. Otherwise, this attribute is ignored and the default for standard output is the console window’s buffer.

hStdError
    If `dwFlags` specifies `STARTF_USESTDHANDLES`, this attribute is the standard error handle for the process. Otherwise, this attribute is ignored and the default for standard error is the console window’s buffer.
The `subprocess` module exposes the following constants.

`subprocess.STD_INPUT_HANDLE`
The standard input device. Initially, this is the console input buffer, CONIN$.

`subprocess.STD_OUTPUT_HANDLE`
The standard output device. Initially, this is the active console screen buffer, CONOUT$.

`subprocess.STD_ERROR_HANDLE`
The standard error device. Initially, this is the active console screen buffer, CONOUT$.

`subprocess.SW_HIDE`
Hides the window. Another window will be activated.

`subprocess.STARTF_USESTDHANDLES`
Specifies that the `STARTUPINFO.hStdInput`, `STARTUPINFO.hStdOutput`, and `STARTUPINFO.hStdError` attributes contain additional information.

`subprocess.STARTF_USESHOWWINDOW`
Specifies that the `STARTUPINFO.wShowWindow` attribute contains
additional information.

subprocess. **CREATE_NEW_CONSOLE**

The new process has a new console, instead of inheriting its parent’s console (the default).

This flag is always set when `Popen` is created with `shell=True`.

subprocess. **CREATE_NEW_PROCESS_GROUP**

A `Popen` `creationflags` parameter to specify that a new process group will be created. This flag is necessary for using `os.kill()` on the subprocess.

This flag is ignored if `CREATE_NEW_CONSOLE` is specified.
17.1.4. Replacing Older Functions with the \texttt{subprocess} Module

In this section, “a becomes b” means that b can be used as a replacement for a.

\textbf{Note:} All “a” functions in this section fail (more or less) silently if the executed program cannot be found; the “b” replacements raise \texttt{OSError} instead.

In addition, the replacements using \texttt{check\_output()} will fail with a \texttt{CalledProcessError} if the requested operation produces a non-zero return code. The output is still available as the \texttt{output} attribute of the raised exception.

In the following examples, we assume that the relevant functions have already been imported from the \texttt{subprocess} module.

17.1.4.1. Replacing /bin/sh shell backquote

```python
output=`mycmd myarg`
```

becomes:

```python
output = check_output(['"mycmd", "myarg"])
```

17.1.4.2. Replacing shell pipeline

```python
output=`dmesg | grep hda`
```

becomes:
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]

The p1.stdout.close() call after starting the p2 is important in order for p1 to receive a SIGPIPE if p2 exits before p1.

Alternatively, for trusted input, the shell's own pipeline support may still be used directly:

```
output=`dmesg | grep hda`
```

becomes:

```
output=check_output("dmesg | grep hda", shell=True)
```

### 17.1.4.3. Replacing `os.system()`

```
status = os.system("mycmd + " myarg")
# becomes
status = subprocess.call("mycmd + " myarg", shell=True)
```

Notes:

- Calling the program through the shell is usually not required.

A more realistic example would look like this:

```
try:
    retcode = call("mycmd + " myarg", shell=True)
    if retcode < 0:
        print >>sys.stderr, "Child was terminated by signal", -
    else:
        print >>sys.stderr, "Child returned", retcode
except OSError as e:
    print >>sys.stderr, "Execution failed:", e
```
17.1.4.4. Replacing the `os.spawn` family

**P_NOWAIT example:**

```python
code
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
>>> pid = Popen(["/bin/mycmd", "myarg"]).pid
```

**P_WAIT example:**

```python
code
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
>>> retcode = call(["/bin/mycmd", "myarg"])
```

**Vector example:**

```python
code
os.spawnvp(os.P_NOWAIT, path, args)
>>> Popen([path] + args[1:])
```

**Environment example:**

```python
code
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
>>> Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

17.1.4.5. Replacing `os.popen()`, `os.popen2()`, `os.popen3()`

```python
code
pipe = os.popen("cmd", 'r', bufsize)
>>> pipe = Popen("cmd", shell=True, bufsize=bufsize, stdout=PIPE).s
```

```python
code
pipe = os.popen("cmd", 'w', bufsize)
>>> pipe = Popen("cmd", shell=True, bufsize=bufsize, stdin=PIPE).s
```
(child_stdin, child_stdout) = os.popen2("cmd", mode, bufsize)
==> 
p = Popen("cmd", shell=True, bufsize=bufsize,
   stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)

(child_stdin, child_stdout, child_stderr) = os.popen3("cmd", mode, bufsize)
==> 
p = Popen("cmd", shell=True, bufsize=bufsize,
   stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin, child_stdout, child_stderr) = (p.stdin, p.stdout, p.stderr)

(child_stdin, child_stdout_and_stderr) = os.popen4("cmd", mode, bufsize)
==> 
p = Popen("cmd", shell=True, bufsize=bufsize,
   stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)

On Unix, os.popen2, os.popen3 and os.popen4 also accept a sequence as the command to execute, in which case arguments will be passed directly to the program without shell intervention. This usage can be replaced as follows:

(child_stdin, child_stdout) = os.popen2(['/bin/ls', '-l'], mode, bufsize)
==> 
p = Popen(['/bin/ls', '-l'], bufsize=bufsize, stdin=PIPE, stdout
(child_stdin, child_stdout) = (p.stdin, p.stdout)

Return code handling translates as follows:

pipe = os.popen("cmd", 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
print "There were some errors"

==> 
process = Popen("cmd", shell=True, stdin=PIPE)
...
process.stdin.close()

if process.wait() != 0:
    print "There were some errors"

17.1.4.6. Replacing functions from the `popen2` module

(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize=

==> 

p = Popen("somestring", shell=True, bufsize=bufsize,
    stdin=PIPE, stdout=PIPE, close_fds=True)

(child_stdout, child_stdin) = (p.stdout, p.stdin)

On Unix, popen2 also accepts a sequence as the command to execute, in which case arguments will be passed directly to the program without shell intervention. This usage can be replaced as follows:

(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"],

==> 

p = Popen(["mycmd", "myarg"], bufsize=bufsize,
    stdin=PIPE, stdout=PIPE, close_fds=True)

(child_stdout, child_stdin) = (p.stdout, p.stdin)

popen2.Popen3 and popen2.Popen4 basically work as subprocess.Popen, except that:

- `Popen` raises an exception if the execution fails.
- the `capturestderr` argument is replaced with the `stderr` argument.
- `stdin=PIPE` and `stdout=PIPE` must be specified.
- popen2 closes all file descriptors by default, but you have to specify `close_fds=True` with `Popen`. 
17.1.5. Notes

17.1.5.1. Converting an argument sequence to a string on Windows

On Windows, an *args sequence is converted to a string that can be parsed using the following rules (which correspond to the rules used by the MS C runtime):

1. Arguments are delimited by white space, which is either a space or a tab.
2. A string surrounded by double quotation marks is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
3. A double quotation mark preceded by a backslash is interpreted as a literal double quotation mark.
4. Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
5. If backslashes immediately precede a double quotation mark, every pair of backslashes is interpreted as a literal backslash. If the number of backslashes is odd, the last backslash escapes the next double quotation mark as described in rule 3.
17.2. socket — Low-level networking interface

This module provides access to the BSD socket interface. It is available on all modern Unix systems, Windows, Mac OS X, BeOS, OS/2, and probably additional platforms.

**Note:** Some behavior may be platform dependent, since calls are made to the operating system socket APIs.

For an introduction to socket programming (in C), see the following papers: An Introductory 4.3BSD Interprocess Communication Tutorial, by Stuart Sechrest and An Advanced 4.3BSD Interprocess Communication Tutorial, by Samuel J. Leffler et al, both in the UNIX Programmer's Manual, Supplementary Documents 1 (sections PS1:7 and PS1:8). The platform-specific reference material for the various socket-related system calls are also a valuable source of information on the details of socket semantics. For Unix, refer to the manual pages; for Windows, see the WinSock (or Winsock 2) specification. For IPv6-ready APIs, readers may want to refer to RFC 3493 titled Basic Socket Interface Extensions for IPv6.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python's object-oriented style: the `socket()` function returns a *socket object* whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with `read()` and `write()` operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

Socket addresses are represented as follows: A single string is used
for the `AF_UNIX` address family. A pair `(host, port)` is used for the `AF_INET` address family, where `host` is a string representing either a hostname in Internet domain notation like `daring.cwi.nl` or an IPv4 address like `'100.50.200.5'`, and `port` is an integer. For `AF_INET6` address family, a four-tuple `(host, port, flowinfo, scopeid)` is used, where `flowinfo` and `scopeid` represents `sin6_flowinfo` and `sin6_scope_id` member in `struct sockaddr_in6` in C. For `socket` module methods, `flowinfo` and `scopeid` can be omitted just for backward compatibility. Note, however, omission of `scopeid` can cause problems in manipulating scoped IPv6 addresses. Other address families are currently not supported. The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created.

For IPv4 addresses, two special forms are accepted instead of a host address: the empty string represents `INADDR_ANY`, and the string `<broadcast>` represents `INADDR_BROADCAST`. The behavior is not available for IPv6 for backward compatibility, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.

If you use a hostname in the `host` portion of IPv4/v6 socket address, the program may show a nondeterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in `host` portion.

*New in version 2.5:* AF_NETLINK sockets are represented as pairs `pid, groups`.

*New in version 2.6:* Linux-only support for TIPC is also available using the `AF_TIPC` address family. TIPC is an open, non-IP based
networked protocol designed for use in clustered computer environments. Addresses are represented by a tuple, and the fields depend on the address type. The general tuple form is 

\[
(\text{addr\_type}, v1, v2, v3 [, scope]),
\]

where:

- \textit{addr\_type} is one of \texttt{TIPC\_ADDR\_NAMESEQ}, \texttt{TIPC\_ADDR\_NAME}, or \texttt{TIPC\_ADDR\_ID}.

- \textit{scope} is one of \texttt{TIPC\_ZONE\_SCOPE}, \texttt{TIPC\_CLUSTER\_SCOPE}, and \texttt{TIPC\_NODE\_SCOPE}.

- If \textit{addr\_type} is \texttt{TIPC\_ADDR\_NAME}, then \textit{v1} is the server type, \textit{v2} is the port identifier, and \textit{v3} should be 0.

  If \textit{addr\_type} is \texttt{TIPC\_ADDR\_NAMESEQ}, then \textit{v1} is the server type, \textit{v2} is the lower port number, and \textit{v3} is the upper port number.

  If \textit{addr\_type} is \texttt{TIPC\_ADDR\_ID}, then \textit{v1} is the node, \textit{v2} is the reference, and \textit{v3} should be set to 0.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised; errors related to socket or address semantics raise the error \texttt{socket.error}.

Non-blocking mode is supported through \texttt{setblocking()}. A generalization of this based on timeouts is supported through \texttt{settimeout()}.

The module \texttt{socket} exports the following constants and functions:

\texttt{exception socket.error}

This exception is raised for socket-related errors. The accompanying value is either a string telling what went wrong or a pair \texttt{(errno, string)} representing an error returned by a
system call, similar to the value accompanying os.error. See the module errno, which contains names for the error codes defined by the underlying operating system.

*Changed in version 2.6:* socket.error is now a child class of IOError.

**exception** socket.herror

This exception is raised for address-related errors, i.e. for functions that use h_errno in the C API, including gethostbyname_ex() and gethostbyaddr().

The accompanying value is a pair (h_errno, string) representing an error returned by a library call. string represents the description of h_errno, as returned by the hstrerror() C function.

**exception** socket.gaierror

This exception is raised for address-related errors, for getaddrinfo() and getnameinfo(). The accompanying value is a pair (error, string) representing an error returned by a library call. string represents the description of error, as returned by the gai_strerror() C function. The error value will match one of the EAI_* constants defined in this module.

**exception** socket.timeout

This exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to settimeout(). The accompanying value is a string whose value is currently always “timed out”.

*New in version 2.3.*

socket.AF_UNIX
socket.AF_INET
socket.AF_INET6

These constants represent the address (and protocol) families, used for the first argument to `socket()`. If the `AF_UNIX` constant is not defined then this protocol is unsupported.

socket.SOCK_STREAM
socket.SOCK_DGRAM
socket.SOCK_RAW
socket.SOCK_RDM
socket.SOCK_SEQPACKET

These constants represent the socket types, used for the second argument to `socket()`. (Only `SOCK_STREAM` and `SOCK_DGRAM` appear to be generally useful.)

SO_*
socket.SOMAXCONN
MSG_*
SOL_*
IPPROTO_*
IADDR_*
IP_*
IPV6_*
EAI_*
AI_*
NI_*
TCP_*

Many constants of these forms, documented in the Unix documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the `setsockopt()` and `getsockopt()` methods of socket objects. In most cases, only those symbols that are defined in the Unix header files are defined; for a few symbols, default values are provided.

SIO_*
RCVALL_*

Constants for Windows' WSAIoctl(). The constants are used as
arguments to the `ioctl()` method of socket objects.

*New in version 2.6.*

**TIPC_*

TIPC related constants, matching the ones exported by the C socket API. See the TIPC documentation for more information.

*New in version 2.6.*

**socket.** has_ipv6

This constant contains a boolean value which indicates if IPv6 is supported on this platform.

*New in version 2.3.*

**socket.** create_connection(address[, timeout[, source_address]])

Connect to a TCP service listening on the Internet address (a 2-tuple `host, port`), and return the socket object. This is a higher-level function than `socket.connect()`: if `host` is a non-numeric hostname, it will try to resolve it for both `AF_INET` and `AF_INET6`, and then try to connect to all possible addresses in turn until a connection succeeds. This makes it easy to write clients that are compatible to both IPv4 and IPv6.

Passing the optional `timeout` parameter will set the timeout on the socket instance before attempting to connect. If no `timeout` is supplied, the global default timeout setting returned by `getdefaulttimeout()` is used.

If supplied, `source_address` must be a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting. If host or port are `` or 0 respectively the OS default behavior will be used.
New in version 2.6.

Changed in version 2.7: source_address was added.

socket.getaddrinfo(host, port[, family[, socktype[, proto[, flags]]]]))

Translate the host/port argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. host is a domain name, a string representation of an IPv4/v6 address or None. port is a string service name such as 'http', a numeric port number or None. By passing None as the value of host and port, you can pass NULL to the underlying C API.

The family, socktype and proto arguments can be optionally specified in order to narrow the list of addresses returned. By default, their value is 0, meaning that the full range of results is selected. The flags argument can be one or several of the AI_ constants, and will influence how results are computed and returned. Its default value is 0. For example, AI_NUMERICHOST will disable domain name resolution and will raise an error if host is a domain name.

The function returns a list of 5-tuples with the following structure:

(family, socktype, proto, canonname, sockaddr)

In these tuples, family, socktype, proto are all integers and are meant to be passed to the socket() function. canonname will be a string representing the canonical name of the host if AI_CANONNAME is part of the flags argument; else canonname will be empty. sockaddr is a tuple describing a socket address, whose format depends on the returned family (a (address, port) 2-tuple for AF_INET, a (address, port, flow info, scope id) 4-
tuple for `AF_INET6`), and is meant to be passed to the `socket.connect()` method.

The following example fetches address information for a hypothetical TCP connection to `example.org` on port 80 (results may differ on your system if IPv6 isn't enabled):

```python
>>> socket.getaddrinfo("example.org", 80, 0, 0, socket.IPPROTO_TCP)
[(10, 1, 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0)),
 (2, 1, 6, '', ('93.184.216.34', 80))]
```

*New in version 2.2.*

socket. `getfqdn([name])`
Return a fully qualified domain name for `name`. If `name` is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by `gethostbyaddr()` is checked, followed by aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available, the hostname as returned by `gethostname()` is returned.

*New in version 2.0.*

socket. `gethostbyname(hostname)`
Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as `'100.50.200.5'`. If the host name is an IPv4 address itself it is returned unchanged. See `gethostbyname_ex()` for a more complete interface. `gethostbyname()` does not support IPv6 name resolution, and `getaddrinfo()` should be used instead for IPv4/v6 dual stack support.

socket. `gethostbyname_ex(hostname)`
Translate a host name to IPv4 address format, extended interface. Return a triple \( (\text{hostname}, \text{aliaslist}, \text{ipaddrlist}) \) where \text{hostname} is the primary host name responding to the given \text{ip_address}, \text{aliaslist} is a (possibly empty) list of alternative host names for the same address, and \text{ipaddrlist} is a list of IPv4 addresses for the same interface on the same host (often but not always a single address). \text{gethostbyname\_ex()} does not support IPv6 name resolution, and \text{getaddrinfo()} should be used instead for IPv4/v6 dual stack support.

\text{socket.\text{gethostname}()}\)

Return a string containing the hostname of the machine where the Python interpreter is currently executing.

If you want to know the current machine's IP address, you may want to use \text{gethostbyname(gethostname())}. This operation assumes that there is a valid address-to-host mapping for the host, and the assumption does not always hold.

Note: \text{gethostname()} doesn't always return the fully qualified domain name; use \text{getfqdn()} (see above).

\text{socket.\text{gethostbyaddr}(ip\_address)}\)

Return a triple \( (\text{hostname}, \text{aliaslist}, \text{ipaddrlist}) \) where \text{hostname} is the primary host name responding to the given \text{ip_address}, \text{aliaslist} is a (possibly empty) list of alternative host names for the same address, and \text{ipaddrlist} is a list of IPv4/v6 addresses for the same interface on the same host (most likely containing only a single address). To find the fully qualified domain name, use the function \text{getfqdn()}. \text{gethostbyaddr()} supports both IPv4 and IPv6.

\text{socket.\text{getnameinfo}(sockaddr, flags)}\)

Translate a socket address \text{sockaddr} into a 2-tuple \( (\text{host, port}) \).
Depending on the settings of flags, the result can contain a fully-qualified domain name or numeric address representation in host. Similarly, port can contain a string port name or a numeric port number.

*New in version 2.2.*

socket. **getprotobynamel** *(protocollname)*

Translate an Internet protocol name (for example, 'icmp') to a constant suitable for passing as the (optional) third argument to the socket() function. This is usually only needed for sockets opened in “raw” mode (SOCK_RAW); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.

socket. **getservbyname** *(servicename[, protocollname]*)

Translate an Internet service name and protocol name to a port number for that service. The optional protocol name, if given, should be 'tcp' or 'udp', otherwise any protocol will match.

socket. **getservbyport** *(port[, protocollname]*)

Translate an Internet port number and protocol name to a service name for that service. The optional protocol name, if given, should be 'tcp' or 'udp', otherwise any protocol will match.

socket. **socket** *(family[, type[, proto]])*

Create a new socket using the given address family, socket type and protocol number. The address family should be AF_INET (the default), AF_INET6 or AF_UNIX. The socket type should be SOCK_STREAM (the default), SOCK_DGRAM or perhaps one of the other sock_ constants. The protocol number is usually zero and may be omitted in that case.
socket.socketpair([family[, type[, proto]]])

Build a pair of connected socket objects using the given address family, socket type, and protocol number. Address family, socket type, and protocol number are as for the socket() function above. The default family is AF_UNIX if defined on the platform; otherwise, the default is AF_INET. Availability: Unix.

*New in version 2.4.*

socket.fromfd(fd, family, type[, proto])

Duplicate the file descriptor fd (an integer as returned by a file object's fileno() method) and build a socket object from the result. Address family, socket type and protocol number are as for the socket() function above. The file descriptor should refer to a socket, but this is not checked — subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (such as a server started by the Unix inet daemon). The socket is assumed to be in blocking mode. Availability: Unix.

socket.ntohl(x)

Convert 32-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.ntohs(x)

Convert 16-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

socket.htonl(x)
Convert 32-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket. **htons**(*x*)

Convert 16-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

socket. **inet_aton**(ip_string)

Convert an IPv4 address from dotted-quad string format (for example, ‘123.45.67.89’) to 32-bit packed binary format, as a string four characters in length. This is useful when conversing with a program that uses the standard C library and needs objects of type **struct in_addr**, which is the C type for the 32-bit packed binary this function returns.

**inet_aton()** also accepts strings with less than three dots; see the Unix manual page **inet(3)** for details.

If the IPv4 address string passed to this function is invalid, socket.error will be raised. Note that exactly what is valid depends on the underlying C implementation of **inet_aton()**.

**inet_aton()** does not support IPv6, and **inet_pton()** should be used instead for IPv4/v6 dual stack support.

socket. **inet_ntoa**(packed_ip)

Convert a 32-bit packed IPv4 address (a string four characters in length) to its standard dotted-quad string representation (for example, ‘123.45.67.89’). This is useful when conversing with a program that uses the standard C library and needs objects of type **struct in_addr**, which is the C type for the 32-bit packed
binary data this function takes as an argument.

If the string passed to this function is not exactly 4 bytes in length, `socket.error` will be raised. `inet_ntoa()` does not support IPv6, and `inet_ntop()` should be used instead for IPv4/v6 dual stack support.

```python
socket.inet_pton(address_family, ip_string)
```
Convert an IP address from its family-specific string format to a packed, binary format. `inet_pton()` is useful when a library or network protocol calls for an object of type `struct in_addr` (similar to `inet_aton()` or `struct in6_addr`).

Supported values for `address_family` are currently `AF_INET` and `AF_INET6`. If the IP address string `ip_string` is invalid, `socket.error` will be raised. Note that exactly what is valid depends on both the value of `address_family` and the underlying implementation of `inet_pton()`.

Availability: Unix (maybe not all platforms).

*New in version 2.3.*

```python
socket.inet_ntop(address_family, packed_ip)
```
Convert a packed IP address (a string of some number of characters) to its standard, family-specific string representation (for example, '7.10.0.5' or '5aef:2b::8') `inet_ntop()` is useful when a library or network protocol returns an object of type `struct in_addr` (similar to `inet_ntoa()`) or `struct in6_addr`.

Supported values for `address_family` are currently `AF_INET` and `AF_INET6`. If the string `packed_ip` is not the correct length for the specified address family, `ValueError` will be raised. A `socket.error` is raised for errors from the call to `inet_ntop()`.
Availability: Unix (maybe not all platforms).

*New in version 2.3.*

**socket.``getdefaulttimeout``()**

Return the default timeout in seconds (float) for new socket objects. A value of `None` indicates that new socket objects have no timeout. When the socket module is first imported, the default is `None`.

*New in version 2.3.*

**socket.``setdefaulttimeout``(timeout)**

Set the default timeout in seconds (float) for new socket objects. A value of `None` indicates that new socket objects have no timeout. When the socket module is first imported, the default is `None`.

*New in version 2.3.*

**socket.``SocketType``**

This is a Python type object that represents the socket object type. It is the same as `type(socket(...))`.

**See also:**

**Module``SocketServer``**

Classes that simplify writing network servers.

**Module``ssl``**

A TLS/SSL wrapper for socket objects.
17.2.1. Socket Objects

Socket objects have the following methods. Except for `makefile()` these correspond to Unix system calls applicable to sockets.

**socket.** `accept()`

Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair `(conn, address)` where `conn` is a new socket object usable to send and receive data on the connection, and `address` is the address bound to the socket on the other end of the connection.

**socket.** `bind(address)`

Bind the socket to `address`. The socket must not already be bound. (The format of `address` depends on the address family — see above.)

**Note:** This method has historically accepted a pair of parameters for `AF_INET` addresses instead of only a tuple. This was never intentional and is no longer available in Python 2.0 and later.

**socket.** `close()`

Close the socket. All future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

**Note:** `close()` releases the resource associated with a connection but does not necessarily close the connection immediately. If you want to close the connection in a timely fashion, call `shutdown()` before `close()`.
socket.**connect**(address)

Connect to a remote socket at *address*. (The format of *address* depends on the address family — see above.)

**Note:** This method has historically accepted a pair of parameters for **AF_INET** addresses instead of only a tuple. This was never intentional and is no longer available in Python 2.0 and later.

socket.**connect_ex**(address)

Like **connect**(address), but return an error indicator instead of raising an exception for errors returned by the C-level **connect**() call (other problems, such as “host not found,” can still raise exceptions). The error indicator is 0 if the operation succeeded, otherwise the value of the **errno** variable. This is useful to support, for example, asynchronous connects.

**Note:** This method has historically accepted a pair of parameters for **AF_INET** addresses instead of only a tuple. This was never intentional and is no longer available in Python 2.0 and later.

socket.**fileno**()

Return the socket’s file descriptor (a small integer). This is useful with **select.select()**.

Under Windows the small integer returned by this method cannot be used where a file descriptor can be used (such as **os.fdopen()**). Unix does not have this limitation.

socket.**getpeername**()

Return the remote address to which the socket is connected. This is useful to find out the port number of a remote IPv4/v6 socket,
for instance. (The format of the address returned depends on the address family — see above.) On some systems this function is not supported.

socket. **getsockname()**

Return the socket's own address. This is useful to find out the port number of an IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.)

socket. **getsockopt**(level, optname[, buflen])

Return the value of the given socket option (see the Unix man page *getsockopt*(2)). The needed symbolic constants (so_* etc.) are defined in this module. If buflen is absent, an integer option is assumed and its integer value is returned by the function. If buflen is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a string. It is up to the caller to decode the contents of the buffer (see the optional built-in module **struct** for a way to decode C structures encoded as strings).

socket. **ioctl**(control, option)

**Platform:** Windows

The **ioctl()** method is a limited interface to the WSAIoctl system interface. Please refer to the Win32 documentation for more information.

On other platforms, the generic **fcntl.fcntl()** and **fcntl.ioctl()** functions may be used; they accept a socket object as their first argument.

*New in version 2.6.*

socket. **listen**(backlog)

Listen for connections made to the socket. The **backlog** argument
specifies the maximum number of queued connections and should be at least 0; the maximum value is system-dependent (usually 5), the minimum value is forced to 0.

socket.**makefile**(["mode", "bufsize"])

Return a *file object* associated with the socket. (File objects are described in File Objects.) The file object does not close the socket explicitly when its *close()* method is called, but only removes its reference to the socket object, so that the socket will be closed if it is not referenced from anywhere else.

The socket must be in blocking mode (it can not have a timeout). The optional *mode* and *bufsize* arguments are interpreted the same way as by the built-in *file()* function.

```
Note: On Windows, the file-like object created by **makefile**() cannot be used where a file object with a file descriptor is expected, such as the stream arguments of subprocess.Popen().
```

socket.**recv**(*bufsize[, flags]*)

Receive data from the socket. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by *bufsize*. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero.

```
Note: For best match with hardware and network realities, the value of *bufsize* should be a relatively small power of 2, for example, 4096.
```

socket.**recvfrom**(*bufsize[, flags]*)

Receive data from the socket. The return value is a pair (*string,*
address) where string is a string representing the data received and address is the address of the socket sending the data. See the Unix manual page recv(2) for the meaning of the optional argument flags; it defaults to zero. (The format of address depends on the address family — see above.)

socket.recvfrom_into(buffer[, nbytes[, flags]])
Receive data from the socket, writing it into buffer instead of creating a new string. The return value is a pair (nbytes, address) where nbytes is the number of bytes received and address is the address of the socket sending the data. See the Unix manual page recv(2) for the meaning of the optional argument flags; it defaults to zero. (The format of address depends on the address family — see above.)

*New in version 2.5.*

socket.recv_into(buffer[, nbytes[, flags]])
Receive up to nbytes bytes from the socket, storing the data into a buffer rather than creating a new string. If nbytes is not specified (or 0), receive up to the size available in the given buffer. Returns the number of bytes received. See the Unix manual page recv(2) for the meaning of the optional argument flags; it defaults to zero.

*New in version 2.5.*

socket.send(string[, flags])
Send data to the socket. The socket must be connected to a remote socket. The optional flags argument has the same meaning as for recv() above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data. For further
information on this concept, consult the Socket Programming HOWTO.

socket. **sendall**(string[, flags])

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for *recv()* above. Unlike *send()* this method continues to send data from *string* until either all data has been sent or an error occurs. **None** is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

socket. **sendto**(string, address)
socket. **sendto**(string, flags, address)

Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*. The optional *flags* argument has the same meaning as for *recv()* above. Return the number of bytes sent. (The format of *address* depends on the address family — see above.)

socket. **setblocking**(flag)

Set blocking or non-blocking mode of the socket: if *flag* is 0, the socket is set to non-blocking, else to blocking mode. Initially all sockets are in blocking mode. In non-blocking mode, if a *recv()* call doesn’t find any data, or if a *send()* call can’t immediately dispose of the data, an **error** exception is raised; in blocking mode, the calls block until they can proceed. *s.setblocking(0)* is equivalent to *s.settimeout(0.0)*; *s.setblocking(1)* is equivalent to *s.settimeout(None)*.

socket. **settimeout**(value)

Set a timeout on blocking socket operations. The *value* argument can be a nonnegative float expressing seconds, or **None**. If a float
is given, subsequent socket operations will raise a timeout exception if the timeout period value has elapsed before the operation has completed. Setting a timeout of None disables timeouts on socket operations. \texttt{s.settimeout}(0.0) is equivalent to \texttt{s.setblocking}(0); \texttt{s.settimeout} (None) is equivalent to \texttt{s.setblocking}(1).

\textit{New in version 2.3.}

\texttt{socket.gettimeout()}  
Return the timeout in seconds (float) associated with socket operations, or None if no timeout is set. This reflects the last call to \texttt{setblocking()} or \texttt{settimeout()}.

\textit{New in version 2.3.}

Some notes on socket blocking and timeouts: A socket object can be in one of three modes: blocking, non-blocking, or timeout. Sockets are always created in blocking mode. In blocking mode, operations block until complete or the system returns an error (such as connection timed out). In non-blocking mode, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately. In timeout mode, operations fail if they cannot be completed within the timeout specified for the socket or if the system returns an error. The \texttt{setblocking()} method is simply a shorthand for certain \texttt{settimeout()} calls.

Timeout mode internally sets the socket in non-blocking mode. The blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. A consequence of this is that file objects returned by the \texttt{makefile()} method must only be used when the socket is in blocking mode; in timeout or non-blocking mode file operations that cannot be completed immediately will fail.
Note that the `connect()` operation is subject to the timeout setting, and in general it is recommended to call `settimeout()` before calling `connect()` or pass a timeout parameter to `create_connection()`. The system network stack may return a connection timeout error of its own regardless of any Python socket timeout setting.

**socket.**`setsockopt(level, optname, value)`

Set the value of the given socket option (see the Unix manual page `setsockopt(2)`). The needed symbolic constants are defined in the `socket` module (`SO_*` etc.). The value can be an integer or a string representing a buffer. In the latter case it is up to the caller to ensure that the string contains the proper bits (see the optional built-in module `struct` for a way to encode C structures as strings).

**socket.**`shutdown(how)`

Shut down one or both halves of the connection. If `how` is `SHUT_RD`, further receives are disallowed. If `how` is `SHUT_WR`, further sends are disallowed. If `how` is `SHUT_RDWR`, further sends and receives are disallowed. Depending on the platform, shutting down one half of the connection can also close the opposite half (e.g. on Mac OS X, `shutdown(SHUT_WR)` does not allow further reads on the other end of the connection).

Note that there are no methods `read()` or `write()`; use `recv()` and `send()` without `flags` argument instead.

Socket objects also have these (read-only) attributes that correspond to the values given to the `socket` constructor.

**socket.**`family`

The socket family.

*New in version 2.5.*
socket. **type**

The socket type.

*New in version 2.5.*

socket. **proto**

The socket protocol.

*New in version 2.5.*
17.2.2. Example

Here are four minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence `socket()`, `bind()`, `listen()`, `accept()` (possibly repeating the `accept()` to service more than one client), while a client only needs the sequence `socket()`, `connect()`. Also note that the server does not `sendall()`/`recv()` on the socket it is listening on but on the new socket returned by `accept()`.

The first two examples support IPv4 only.

```python
# Echo server program
import socket

HOST = ''  # Symbolic name meaning all available interfaces
PORT = 50007  # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(((HOST, PORT)))
s.listen(1)
conn, addr = s.accept()
print('Connected by', addr)
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.sendall(data)
conn.close()

# Echo client program
import socket

HOST = 'daring.cwi.nl'  # The remote host
PORT = 50007  # The same port as used by the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(((HOST, PORT)))
s.sendall('Hello, world!')
data = s.recv(1024)
s.close()
```
The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to the all addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

```python
print 'Received', repr(data)

HOST = None  # Symbolic name meaning all available
PORT = 50007 # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except socket.error as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print 'could not open socket'
    sys.exit(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()
```
# Echo client program

```python
import socket
import sys

HOST = 'daring.cwi.nl'  # The remote host
PORT = 50007          # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM, 0, socket.AI_ADDRCONFIG):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except socket.error as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print 'could not open socket'
s.sendall('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', repr(data)
```

The last example shows how to write a very simple network sniffer with raw sockets on Windows. The example requires administrator privileges to modify the interface:

```python
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_TCP)
s.bind((HOST, 0))
```
# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a package
print s.recvfrom(65565)

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)

Running an example several times with too small delay between executions, could lead to this error:

```
socket.error: [Errno 98] Address already in use
```

This is because the previous execution has left the socket in a `TIME_WAIT` state, and can’t be immediately reused.

There is a `socket` flag to set, in order to prevent this, `socket.SO_REUSEADDR`:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

the `SO_REUSEADDR` flag tells the kernel to reuse a local socket in `TIME_WAIT` state, without waiting for its natural timeout to expire.
This module provides access to Transport Layer Security (often known as “Secure Sockets Layer”) encryption and peer authentication facilities for network sockets, both client-side and server-side. This module uses the OpenSSL library. It is available on all modern Unix systems, Windows, Mac OS X, and probably additional platforms, as long as OpenSSL is installed on that platform.

**Note:** Some behavior may be platform dependent, since calls are made to the operating system socket APIs. The installed version of OpenSSL may also cause variations in behavior. For example, TLSv1.1 and TLSv1.2 come with openssl version 1.0.1.

**Warning:** Don’t use this module without reading the Security considerations. Doing so may lead to a false sense of security, as the default settings of the ssl module are not necessarily appropriate for your application.

This section documents the objects and functions in the `ssl` module; for more general information about TLS, SSL, and certificates, the reader is referred to the documents in the “See Also” section at the bottom.

This module provides a class, `ssl.SSLSocket`, which is derived from the `socket.socket` type, and provides a socket-like wrapper that also
encrypts and decrypts the data going over the socket with SSL. It supports additional methods such as `getpeercert()`, which retrieves the certificate of the other side of the connection, and `cipher()`, which retrieves the cipher being used for the secure connection.

For more sophisticated applications, the `ssl.SSLContext` class helps manage settings and certificates, which can then be inherited by SSL sockets created through the `SSLContext.wrap_socket()` method.
17.3.1. Functions, Constants, and Exceptions

**exception** ssl. **SSLError**

Raised to signal an error from the underlying SSL implementation (currently provided by the OpenSSL library). This signifies some problem in the higher-level encryption and authentication layer that's superimposed on the underlying network connection. This error is a subtype of **socket.error**, which in turn is a subtype of **IOError**. The error code and message of **SSLError** instances are provided by the OpenSSL library.

**library**

A string mnemonic designating the OpenSSL submodule in which the error occurred, such as **SSL**, **PEM** or **X509**. The range of possible values depends on the OpenSSL version.

*New in version 2.7.9.*

**reason**

A string mnemonic designating the reason this error occurred, for example **CERTIFICATE_VERIFY_FAILED**. The range of possible values depends on the OpenSSL version.

*New in version 2.7.9.*

**exception** ssl. **SSLZeroReturnError**

A subclass of **SSLError** raised when trying to read or write and the SSL connection has been closed cleanly. Note that this doesn’t mean that the underlying transport (read TCP) has been closed.

*New in version 2.7.9.*
exception ssl. **SSLWantReadError**

A subclass of **SSLError** raised by a non-blocking SSL socket when trying to read or write data, but more data needs to be received on the underlying TCP transport before the request can be fulfilled.

*New in version 2.7.9.*

exception ssl. **SSLWantWriteError**

A subclass of **SSLError** raised by a non-blocking SSL socket when trying to read or write data, but more data needs to be sent on the underlying TCP transport before the request can be fulfilled.

*New in version 2.7.9.*

exception ssl. **SSLSyscallError**

A subclass of **SSLError** raised when a system error was encountered while trying to fulfill an operation on a SSL socket. Unfortunately, there is no easy way to inspect the original errno number.

*New in version 2.7.9.*

exception ssl. **SSLEOFError**

A subclass of **SSLError** raised when the SSL connection has been terminated abruptly. Generally, you shouldn’t try to reuse the underlying transport when this error is encountered.

*New in version 2.7.9.*

exception ssl. **CertificateError**

Raised to signal an error with a certificate (such as mismatching hostname). Certificate errors detected by OpenSSL, though, raise an **SSLError**.
17.3.1.1. Socket creation

The following function allows for standalone socket creation. Starting from Python 2.7.9, it can be more flexible to use `SSLContext.wrap_socket()` instead.

```python
ssl.wrap_socket(sock, keyfile=None, certfile=None,
server_side=False, cert_reqs=CERT_NONE, ssl_version={see docs}, ca_certs=None, do_handshake_on_connect=True,
suppress_ragged_eofs=True, ciphers=None)
```

Takes an instance `sock` of `socket.socket`, and returns an instance of `ssl.SSLSocket`, a subtype of `socket.socket`, which wraps the underlying socket in an SSL context. `sock` must be a `SOCK_STREAM` socket; other socket types are unsupported.

For client-side sockets, the context construction is lazy; if the underlying socket isn’t connected yet, the context construction will be performed after `connect()` is called on the socket. For server-side sockets, if the socket has no remote peer, it is assumed to be a listening socket, and the server-side SSL wrapping is automatically performed on client connections accepted via the `accept()` method. `wrap_socket()` may raise `SSLError`.

The `keyfile` and `certfile` parameters specify optional files which contain a certificate to be used to identify the local side of the connection. See the discussion of `Certificates` for more information on how the certificate is stored in the `certfile`.

The parameter `server_side` is a boolean which identifies whether server-side or client-side behavior is desired from this socket.

The parameter `cert_reqs` specifies whether a certificate is required from the other side of the connection, and whether it will
be validated if provided. It must be one of the three values
CERT_NONE (certificates ignored), CERT_OPTIONAL (not required, but
validated if provided), or CERT_REQUIRED (required and validated).
If the value of this parameter is not CERT_NONE, then the ca_certs
parameter must point to a file of CA certificates.

The ca_certs file contains a set of concatenated “certification
authority” certificates, which are used to validate certificates
passed from the other end of the connection. See the discussion
of Certificates for more information about how to arrange the
certificates in this file.

The parameter ssl_version specifies which version of the SSL
protocol to use. Typically, the server chooses a particular protocol
version, and the client must adapt to the server’s choice. Most of
the versions are not interoperable with the other versions. If not
specified, the default is PROTOCOL_SSLv23; it provides the most
compatibility with other versions.

Here’s a table showing which versions in a client (down the side)
can connect to which versions in a server (along the top):

<table>
<thead>
<tr>
<th>client / server</th>
<th>SSLv2</th>
<th>SSLv3</th>
<th>SSLv23</th>
<th>TLSv1</th>
<th>TLSv1.1</th>
<th>TLSv1.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSLv2</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>SSLv3</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>SSLv23</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>TLSv1</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>TLSv1.1</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>TLSv1.2</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Note: Which connections succeed will vary depending on the
version of OpenSSL. For example, before OpenSSL 1.0.0, an
SSLv23 client would always attempt SSLv2 connections.
The ciphers parameter sets the available ciphers for this SSL object. It should be a string in the OpenSSL cipher list format.

The parameter do_handshake_on_connect specifies whether to do the SSL handshake automatically after doing a socket.connect(), or whether the application program will call it explicitly, by invoking the SSLSocket.do_handshake() method. Calling SSLSocket.do_handshake() explicitly gives the program control over the blocking behavior of the socket I/O involved in the handshake.

The parameter suppress_ragged_eofs specifies how the SSLSocket.read() method should signal unexpected EOF from the other end of the connection. If specified as True (the default), it returns a normal EOF (an empty bytes object) in response to unexpected EOF errors raised from the underlying socket; if False, it will raise the exceptions back to the caller.

*Changed in version 2.7:* New optional argument ciphers.

### 17.3.1.2. Context creation

A convenience function helps create SSLContext objects for common purposes.

```python
ssl.create_default_context(purpose=Purpose.SERVER_AUTH, cafile=None, capath=None, cadata=None)
```

Return a new SSLContext object with default settings for the given purpose. The settings are chosen by the ssl module, and usually represent a higher security level than when calling the SSLContext constructor directly.

cafile, capath, cadata represent optional CA certificates to trust for certificate verification, as in
SSLContext.load_verify_locations(). If all three are None, this function can choose to trust the system’s default CA certificates instead.

The settings are: PROTOCOL_SSLv23, OP_NO_SSLv2, and OP_NO_SSLv3 with high encryption cipher suites without RC4 and without unauthenticated cipher suites. Passing SERVER_AUTH as purpose sets verify_mode to CERT_REQUIRED and either loads CA certificates (when at least one of cafile, capath or cadata is given) or uses SSLContext.load_default_certs() to load default CA certificates.

**Note:** The protocol, options, cipher and other settings may change to more restrictive values anytime without prior deprecation. The values represent a fair balance between compatibility and security.
If your application needs specific settings, you should create a SSLContext and apply the settings yourself.

**Note:** If you find that when certain older clients or servers attempt to connect with a SSLContext created by this function that they get an error stating “Protocol or cipher suite mismatch”, it may be that they only support SSL3.0 which this function excludes using the OP_NO_SSLv3. SSL3.0 is widely considered to be completely broken. If you still wish to continue to use this function but still allow SSL 3.0 connections you can re-enable them using:

```python
ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
ctx.options &= ~ssl.OP_NO_SSLv3
```

*New in version 2.7.9.*

*Changed in version 2.7.10:* RC4 was dropped from the default
cipher string.

*Changed in version 2.7.13:* ChaCha20/Poly1305 was added to the default cipher string.

3DES was dropped from the default cipher string.

**ssl.

```python
https_verify_certificates(enable=True)
```

Specifies whether or not server certificates are verified when creating client HTTPS connections without specifying a particular SSL context.

Starting with Python 2.7.9, `httplib` and modules which use it, such as `urllib2` and `xmlrpclib`, default to verifying remote server certificates received when establishing client HTTPS connections. This default verification checks that the certificate is signed by a Certificate Authority in the system trust store and that the Common Name (or Subject Alternate Name) on the presented certificate matches the requested host.

Setting `enable` to `True` ensures this default behaviour is in effect.

Setting `enable` to `False` reverts the default HTTPS certificate handling to that of Python 2.7.8 and earlier, allowing connections to servers using self-signed certificates, servers using certificates signed by a Certicate Authority not present in the system trust store, and servers where the hostname does not match the presented server certificate.

The leading underscore on this function denotes that it intentionally does not exist in any implementation of Python 3 and may not be present in all Python 2.7 implementations. The portable approach to bypassing certificate checks or the system trust store when necessary is for tools to enable that on a case-by-case basis by explicitly passing in a suitably configured SSL
context, rather than reverting the default behaviour of the standard library client modules.

*New in version 2.7.12.*

**See also:**
- CVE-2014-9365 – HTTPS man-in-the-middle attack against Python clients using default settings
- PEP 476 – Enabling certificate verification by default for HTTPS
- PEP 493 – HTTPS verification migration tools for Python 2.7

### 17.3.1.3. Random generation

*Deprecated since version 2.7.13:* OpenSSL has deprecated `ssl.RAND_pseudo_bytes()`, use `ssl.RAND_bytes()` instead.

`ssl.RAND_status()`

Return `True` if the SSL pseudo-random number generator has been seeded with ‘enough’ randomness, and `False` otherwise.

You can use `ssl.RAND_egd()` and `ssl.RAND_add()` to increase the randomness of the pseudo-random number generator.

`ssl.RAND_egd(path)`

If you are running an entropy-gathering daemon (EGD) somewhere, and `path` is the pathname of a socket connection open to it, this will read 256 bytes of randomness from the socket, and add it to the SSL pseudo-random number generator to increase the security of generated secret keys. This is typically only necessary on systems without better sources of randomness.

Availability: not available with LibreSSL and OpenSSL > 1.1.0

ssl.**RAND_add**(bytes, entropy)

Mix the given bytes into the SSL pseudo-random number generator. The parameter entropy (a float) is a lower bound on the entropy contained in string (so you can always use 0.0). See **RFC 1750** for more information on sources of entropy.

### 17.3.1.4. Certificate handling

ssl.**match_hostname**(cert, hostname)

Verify that cert (in decoded format as returned by SSLSocket.getpeercert()) matches the given hostname. The rules applied are those for checking the identity of HTTPS servers as outlined in **RFC 2818** and **RFC 6125**, except that IP addresses are not currently supported. In addition to HTTPS, this function should be suitable for checking the identity of servers in various SSL-based protocols such as FTPS, IMAPS, POPS and others.

**CertificateError** is raised on failure. On success, the function returns nothing:

```python
>>> cert = {'subject': ((('commonName', 'example.com'),),)}
>>> ssl.match_hostname(cert, "example.com")
>>> ssl.match_hostname(cert, "example.org")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/py3k/Lib/ssl.py", line 130, in match_hostname
ssl.CertificateError: hostname 'example.org' doesn't match 'example.com'
```

*New in version 2.7.9.*
ssl.cert_time_to_seconds(cert_time)

Return the time in seconds since the Epoch, given the cert_time string representing the “notBefore” or “notAfter” date from a certificate in "%b %d %H:%M:%S %Y %Z" strftime format (C locale).

Here’s an example:

```python
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan 5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

“notBefore” or “notAfter” dates must use GMT (RFC 5280).

Changed in version 2.7.9: Interpret the input time as a time in UTC as specified by ‘GMT’ timezone in the input string. Local timezone was used previously. Return an integer (no fractions of a second in the input format)

ssl.get_server_certificate(addr, ssl_version=PROTOCOL_SSLv23, ca_certs=None)

Given the address addr of an SSL-protected server, as a (hostname, port-number) pair, fetches the server’s certificate, and returns it as a PEM-encoded string. If ssl_version is specified, uses that version of the SSL protocol to attempt to connect to the server. If ca_certs is specified, it should be a file containing a list of root certificates, the same format as used for the same parameter in wrap_socket(). The call will attempt to validate the server certificate against that set of root certificates, and will fail if the validation attempt fails.

Changed in version 2.7.9: This function is now IPv6-compatible, and the default ssl_version is changed from PROTOCOL_SSLv3 to
PROTOCOL_SSLv3 for maximum compatibility with modern servers.

ssl.DER_cert_to_PEM_cert(DER_cert_bytes)
Given a certificate as a DER-encoded blob of bytes, returns a PEM-encoded string version of the same certificate.

ssl.PEM_cert_to_DER_cert(PEM_cert_string)
Given a certificate as an ASCII PEM string, returns a DER-encoded sequence of bytes for that same certificate.

ssl.get_default_verify_paths()
Returns a named tuple with paths to OpenSSL's default cafile and capath. The paths are the same as used by SSLContext.set_default_verify_paths(). The return value is a named tuple DefaultVerifyPaths:

- **cafile** - resolved path to cafile or **None** if the file doesn't exist,
- **capath** - resolved path to capath or **None** if the directory doesn't exist,
- **openssl_cafile_env** - OpenSSL's environment key that points to a cafile,
- **openssl_cafile** - hard coded path to a cafile,
- **openssl_capath_env** - OpenSSL's environment key that points to a capath,
- **openssl_capath** - hard coded path to a capath directory

Availability: LibreSSL ignores the environment vars openssl_cafile_env and openssl_capath_env

New in version 2.7.9.

ssl.enum_certificates(store_name)
Retrieve certificates from Windows’ system cert store. *store_name* may be one of **CA**, **ROOT** or **MY**. Windows may provide
additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either \texttt{x509\_asn} for X.509 ASN.1 data or \texttt{pkcs\_7\_asn} for PKCS#7 ASN.1 data. Trust specifies the purpose of the certificate as a set of OIDS or exactly \texttt{True} if the certificate is trustworthy for all purposes.

Example:

```python
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509 ASN', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}), (b'data...', 'x509 ASN', True)]
```

Availability: Windows.

\textit{New in version 2.7.9.}

\texttt{ssl.enum\_crls(store\_name)}

Retrieve CRLs from Windows' system cert store. \textit{store\_name} may be one of \texttt{CA}, \texttt{ROOT} or \texttt{MY}. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either \texttt{x509\_asn} for X.509 ASN.1 data or \texttt{pkcs\_7\_asn} for PKCS#7 ASN.1 data.

Availability: Windows.

\textit{New in version 2.7.9.}

17.3.1.5. Constants

\texttt{ssl.CERT\_NONE}
Possible value for `SSLContext.verify_mode`, or the `cert_reqs` parameter to `wrap_socket()`. In this mode (the default), no certificates will be required from the other side of the socket connection. If a certificate is received from the other end, no attempt to validate it is made.

See the discussion of Security considerations below.

**ssl.CERT_OPTIONAL**

Possible value for `SSLContext.verify_mode`, or the `cert_reqs` parameter to `wrap_socket()`. In this mode no certificates will be required from the other side of the socket connection; but if they are provided, validation will be attempted and an `SSLError` will be raised on failure.

Use of this setting requires a valid set of CA certificates to be passed, either to `SSLContext.load_verify_locations()` or as a value of the `ca_certs` parameter to `wrap_socket()`.

**ssl.CERT_REQUIRED**

Possible value for `SSLContext.verify_mode`, or the `cert_reqs` parameter to `wrap_socket()`. In this mode, certificates are required from the other side of the socket connection; an `SSLError` will be raised if no certificate is provided, or if its validation fails.

Use of this setting requires a valid set of CA certificates to be passed, either to `SSLContext.load_verify_locations()` or as a value of the `ca_certs` parameter to `wrap_socket()`.

**ssl.VERIFY_DEFAULT**

Possible value for `SSLContext.verify_flags`. In this mode, certificate revocation lists (CRLs) are not checked. By default OpenSSL does neither require nor verify CRLs.
New in version 2.7.9.

ssl. **VERIFY_CRL_CHECK LEAF**
Possible value for `SSLContext.verify_flags`. In this mode, only the peer cert is check but non of the intermediate CA certificates. The mode requires a valid CRL that is signed by the peer cert's issuer (its direct ancestor CA). If no proper has been loaded `SSLContext.load_verify_locations`, validation will fail.

New in version 2.7.9.

ssl. **VERIFY_CRL_CHECK_CHAIN**
Possible value for `SSLContext.verify_flags`. In this mode, CRLs of all certificates in the peer cert chain are checked.

New in version 2.7.9.

ssl. **VERIFY_X509 STRICT**
Possible value for `SSLContext.verify_flags` to disable workarounds for broken X.509 certificates.

New in version 2.7.9.

ssl. **VERIFY_X509 TRUSTED FIRST**
Possible value for `SSLContext.verify_flags`. It instructs OpenSSL to prefer trusted certificates when building the trust chain to validate a certificate. This flag is enabled by default.

New in version 2.7.10.

ssl. **PROTOCOL_TLS**
Selects the highest protocol version that both the client and server support. Despite the name, this option can select “TLS” protocols as well as “SSL”.

New in version 2.7.13.
ssl. **PROTOCOL_SSLv23**

Alias for **PROTOCOL_TLS**.

*Deprecated since version 2.7.13: Use **PROTOCOL_TLS** instead.*

ssl. **PROTOCOL_SSLv2**

Selects SSL version 2 as the channel encryption protocol.

This protocol is not available if OpenSSL is compiled with the **OPENSSL_NO_SSL2** flag.

**Warning:** SSL version 2 is insecure. Its use is highly discouraged.

*Deprecated since version 2.7.13: OpenSSL has removed support for SSLv2.*

ssl. **PROTOCOL_SSLv3**

Selects SSL version 3 as the channel encryption protocol.

This protocol is not be available if OpenSSL is compiled with the **OPENSSL_NO_SSLv3** flag.

**Warning:** SSL version 3 is insecure. Its use is highly discouraged.

*Deprecated since version 2.7.13: OpenSSL has deprecated all version specific protocols. Use the default protocol with flags like **OP_NO_SSLv3** instead.*

ssl. **PROTOCOL_TLSv1**

Selects TLS version 1.0 as the channel encryption protocol.

*Deprecated since version 2.7.13: OpenSSL has deprecated all
version specific protocols. Use the default protocol with flags like `OP_NO_SSLv3` instead.

**ssl. PROTOCOL_TLSv1_1**
Selects TLS version 1.1 as the channel encryption protocol. Available only with openssl version 1.0.1+.

*New in version 2.7.9.*

*Deprecated since version 2.7.13:* OpenSSL has deprecated all version specific protocols. Use the default protocol with flags like `OP_NO_SSLv3` instead.

**ssl. PROTOCOL_TLSv1_2**
Selects TLS version 1.2 as the channel encryption protocol. This is the most modern version, and probably the best choice for maximum protection, if both sides can speak it. Available only with openssl version 1.0.1+.

*New in version 2.7.9.*

*Deprecated since version 2.7.13:* OpenSSL has deprecated all version specific protocols. Use the default protocol with flags like `OP_NO_SSLv3` instead.

**ssl. OP_ALL**
Enables workarounds for various bugs present in other SSL implementations. This option is set by default. It does not necessarily set the same flags as OpenSSL’s `SSL_OP_ALL` constant.

*New in version 2.7.9.*

**ssl. OP_NO_SSLv2**
Prevents an SSLv2 connection. This option is only applicable in
conjunction with PROTOCOL_SSLv23. It prevents the peers from choosing SSLv2 as the protocol version.

New in version 2.7.9.

ssl. OP_NO_SSLv3
Prevents an SSLv3 connection. This option is only applicable in conjunction with PROTOCOL_SSLv23. It prevents the peers from choosing SSLv3 as the protocol version.

New in version 2.7.9.

ssl. OP_NO_TLSv1
Prevents a TLSv1 connection. This option is only applicable in conjunction with PROTOCOL_SSLv23. It prevents the peers from choosing TLSv1 as the protocol version.

New in version 2.7.9.

ssl. OP_NO_TLSv1_1
Prevents a TLSv1.1 connection. This option is only applicable in conjunction with PROTOCOL_SSLv23. It prevents the peers from choosing TLSv1.1 as the protocol version. Available only with openssl version 1.0.1+.

New in version 2.7.9.

ssl. OP_NO_TLSv1_2
Prevents a TLSv1.2 connection. This option is only applicable in conjunction with PROTOCOL_SSLv23. It prevents the peers from choosing TLSv1.2 as the protocol version. Available only with openssl version 1.0.1+.

New in version 2.7.9.

ssl. OP_CIPHER_SERVER_PREFERENCE
Use the server’s cipher ordering preference, rather than the client’s. This option has no effect on client sockets and SSLv2 server sockets.

*New in version 2.7.9.*

ssl. **OP_SINGLE_DH_USE**

Prevents re-use of the same DH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

*New in version 2.7.9.*

ssl. **OP_SINGLE_ECDH_USE**

Prevents re-use of the same ECDH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

*New in version 2.7.9.*

ssl. **OP_NO_COMPRESSION**

Disable compression on the SSL channel. This is useful if the application protocol supports its own compression scheme.

This option is only available with OpenSSL 1.0.0 and later.

*New in version 2.7.9.*

ssl. **HAS_ALPN**

Whether the OpenSSL library has built-in support for the Application-Layer Protocol Negotiation TLS extension as described in [RFC 7301](https://tools.ietf.org/html/rfc7301).

*New in version 2.7.10.*

ssl. **HAS_ECDH**

Whether the OpenSSL library has built-in support for Elliptic
Curve-based Diffie-Hellman key exchange. This should be true unless the feature was explicitly disabled by the distributor.

*New in version 2.7.9.*

**ssl. HAS_SNI**

Whether the OpenSSL library has built-in support for the *Server Name Indication* extension (as defined in [RFC 4366](https://tools.ietf.org/html/rfc4366)).

*New in version 2.7.9.*

**ssl. HAS_NPN**

Whether the OpenSSL library has built-in support for *Next Protocol Negotiation* as described in the [NPN draft specification](https://tools.ietf.org/html/draft-ietf-tls-npn-08). When true, you can use the `SSLContext.set_npn_protocols()` method to advertise which protocols you want to support.

*New in version 2.7.9.*

**ssl. CHANNEL_BINDING_TYPES**

List of supported TLS channel binding types. Strings in this list can be used as arguments to `SSLSocket.get_channel_binding()`.

*New in version 2.7.9.*

**ssl. OPENSSL_VERSION**

The version string of the OpenSSL library loaded by the interpreter:

```python
>>> ssl.OPENSSL_VERSION
'OpenSSL 0.9.8k 25 Mar 2009'
```

*New in version 2.7.*

**ssl. OPENSSL_VERSION_INFO**

A tuple of five integers representing version information about the OpenSSL library:
ssl.OPENSSL_VERSION_INFO
(0, 9, 8, 11, 15)

New in version 2.7.

ssl.OPENSSL_VERSION_NUMBER
The raw version number of the OpenSSL library, as a single integer:

```python
〉〉〉 ssl.OPENSSL_VERSION_NUMBER
9470143L
〉〉〉 hex(ssl.OPENSSL_VERSION_NUMBER)
'0x9080bfL'
```

New in version 2.7.

ssl.ALERT_DESCRIPTION_HANDSHAKE_FAILURE
ssl.ALERT_DESCRIPTION_INTERNAL_ERROR
ssl.ALERT_DESCRIPTION_*
Alert Descriptions from RFC 5246 and others. The IANA TLS Alert Registry contains this list and references to the RFCs where their meaning is defined.

Used as the return value of the callback function in SSLContext.set_servername_callback().

New in version 2.7.9.

Purpose. SERVER_AUTH
Option for create_default_context() and SSLContext.load_default_certs(). This value indicates that the context may be used to authenticate Web servers (therefore, it will be used to create client-side sockets).

New in version 2.7.9.

Purpose. CLIENT_AUTH
Option for create_default_context() and
SSLContext.load_default_certs(). This value indicates that the context may be used to authenticate Web clients (therefore, it will be used to create server-side sockets).

New in version 2.7.9.
17.3.2. SSL Sockets

SSL sockets provide the following methods of Socket Objects:

- `accept()`
- `bind()`
- `close()`
- `connect()`
- `fileno()`
- `getpeername()`, `getsockname()`
- `getsockopt()`, `setsockopt()`
- `gettimeout()`, `settimeout()`, `setblocking()`
- `listen()`
- `makefile()`
- `recv()`, `recv_into()` (but passing a non-zero flags argument is not allowed)
- `send()`, `sendall()` (with the same limitation)
- `shutdown()`

However, since the SSL (and TLS) protocol has its own framing atop of TCP, the SSL sockets abstraction can, in certain respects, diverge from the specification of normal, OS-level sockets. See especially the notes on non-blocking sockets.

SSL sockets also have the following additional methods and attributes:

`SSLSocket` . **do_handshake()**

Perform the SSL setup handshake.

*Changed in version 2.7.9:* The handshake method also performs `match_hostname()` when the `check_hostname` attribute of the
socket's `context` is true.

`SSLSocket.getpeercert(binary_form=False)`

If there is no certificate for the peer on the other end of the connection, return `None`. If the SSL handshake hasn’t been done yet, raise `ValueError`.

If the `binary_form` parameter is `False`, and a certificate was received from the peer, this method returns a `dict` instance. If the certificate was not validated, the dict is empty. If the certificate was validated, it returns a dict with several keys, amongst them `subject` (the principal for which the certificate was issued) and `issuer` (the principal issuing the certificate). If a certificate contains an instance of the `Subject Alternative Name` extension (see RFC 3280), there will also be a `subjectAltName` key in the dictionary.

The `subject` and `issuer` fields are tuples containing the sequence of relative distinguished names (RDNs) given in the certificate’s data structure for the respective fields, and each RDN is a sequence of name-value pairs. Here is a real-world example:

```python
{'issuer': ((({'countryName': 'IL'},),),
 (({'organizationName': 'StartCom Ltd.'},),
 (({'organizationalUnitName': 'Secure Digital Certificate Signing'},),
 (({'commonName': 'StartCom Class 2 Primary Intermediate Server'},),
 'notAfter': 'Nov 22 08:15:19 2013 GMT',
 'notBefore': 'Nov 21 03:09:52 2011 GMT',
 'serialNumber': '95F0',
 'subject': ((({'description': '571208-SLe257oHY9fVQ07Z'},),
 (({'countryName': 'US'},),
 (({'stateOrProvinceName': 'California'},),
 (({'localityName': 'San Francisco'},),
 (({'organizationName': 'Electronic Frontier Foundation'},),
 (({'commonName': '*.eff.org'},),
```
((('emailAddress', 'hostmaster@eff.org'),),)
'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org'))
'version': 3

**Note:** To validate a certificate for a particular service, you can use the `match_hostname()` function.

If the `binary_form` parameter is `True`, and a certificate was provided, this method returns the DER-encoded form of the entire certificate as a sequence of bytes, or `None` if the peer did not provide a certificate. Whether the peer provides a certificate depends on the SSL socket's role:

- for a client SSL socket, the server will always provide a certificate, regardless of whether validation was required;
- for a server SSL socket, the client will only provide a certificate when requested by the server; therefore `getpeercert()` will return `None` if you used `CERT_NONE` (rather than `CERT OPTIONAL` or `CERT_REQUIRED`).

*Changed in version 2.7.9:* The returned dictionary includes additional items such as `issuer` and `notBefore`. Additionally, `ValueError` is raised when the handshake isn't done. The returned dictionary includes additional X509v3 extension items such as `crlDistributionPoints`, `caIssuers` and `OCSP URIs`.

**SSLSocket.cipher()**

Returns a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used. If no connection has been established, returns `None`.

**SSLSocket.compression()**

Return the compression algorithm being used as a string, or `None` if the connection isn't compressed.
If the higher-level protocol supports its own compression mechanism, you can use `OP_NO_COMPRESSION` to disable SSL-level compression.

*New in version 2.7.9.*

```python
SSLSocket.get_channel_binding(cb_type="tls-unique")
```

Get channel binding data for current connection, as a bytes object. Returns `None` if not connected or the handshake has not been completed.

The `cb_type` parameter allow selection of the desired channel binding type. Valid channel binding types are listed in the `CHANNEL_BINDING_TYPES` list. Currently only the 'tls-unique' channel binding, defined by [RFC 5929](https://tools.ietf.org/html/rfc5929), is supported. `ValueError` will be raised if an unsupported channel binding type is requested.

*New in version 2.7.9.*

```python
SSLSocket.selected_alpn_protocol()
```

Return the protocol that was selected during the TLS handshake. If `SSLContext.set_alpn_protocols()` was not called, if the other party does not support ALPN, if this socket does not support any of the client's proposed protocols, or if the handshake has not happened yet, `None` is returned.

*New in version 2.7.10.*

```python
SSLSocket.selected_npn_protocol()
```

Return the higher-level protocol that was selected during the TLS/SSL handshake. If `SSLContext.set_npn_protocols()` was not called, or if the other party does not support NPN, or if the handshake has not yet happened, this will return `None`.

*New in version 2.7.9.*
SSLSocket. **unwrap()**
Performs the SSL shutdown handshake, which removes the TLS layer from the underlying socket, and returns the underlying socket object. This can be used to go from encrypted operation over a connection to unencrypted. The returned socket should always be used for further communication with the other side of the connection, rather than the original socket.

SSLSocket. **version()**
Return the actual SSL protocol version negotiated by the connection as a string, or None is no secure connection is established. As of this writing, possible return values include "SSLv2", "SSLv3", "TLSv1", "TLSv1.1" and "TLSv1.2". Recent OpenSSL versions may define more return values.

*New in version 2.7.9.*

SSLSocket. **context**
The SSLContext object this SSL socket is tied to. If the SSL socket was created using the top-level wrap_socket() function (rather than SSLContext.wrap_socket()), this is a custom context object created for this SSL socket.

*New in version 2.7.9.*
17.3.3. SSL Contexts

New in version 2.7.9.

An SSL context holds various data longer-lived than single SSL connections, such as SSL configuration options, certificate(s) and private key(s). It also manages a cache of SSL sessions for server-side sockets, in order to speed up repeated connections from the same clients.

```python
class ssl.SSLContext(protocol)
    Create a new SSL context. You must pass protocol which must be one of the PROTOCOL_* constants defined in this module. PROTOCOL_SSLv23 is currently recommended for maximum interoperability.
```

See also: create_default_context() lets the ssl module choose security settings for a given purpose.

SSLContext objects have the following methods and attributes:

SSLContext.cert_store_stats()
    Get statistics about quantities of loaded X.509 certificates, count of X.509 certificates flagged as CA certificates and certificate revocation lists as dictionary.

Example for a context with one CA cert and one other cert:

```python
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

SSLContext.load_cert_chain(certfile, keyfile=None, password=None)
    Load a private key and the corresponding certificate. The certfile
string must be the path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate’s authenticity. The *keyfile* string, if present, must point to a file containing the private key in. Otherwise the private key will be taken from *certfile* as well. See the discussion of *Certificates* for more information on how the certificate is stored in the *certfile*.

The *password* argument may be a function to call to get the password for decrypting the private key. It will only be called if the private key is encrypted and a password is necessary. It will be called with no arguments, and it should return a string, bytes, or bytearray. If the return value is a string it will be encoded as UTF-8 before using it to decrypt the key. Alternatively a string, bytes, or bytearray value may be supplied directly as the *password* argument. It will be ignored if the private key is not encrypted and no password is needed.

If the *password* argument is not specified and a password is required, OpenSSL’s built-in password prompting mechanism will be used to interactively prompt the user for a password.

An *SSLError* is raised if the private key doesn’t match with the certificate.

**SSLContext.load_default_certs**(*purpose=Purpose.SERVER_AUTH*)

Load a set of default “certification authority” (CA) certificates from default locations. On Windows it loads CA certs from the *CA* and *ROOT* system stores. On other systems it calls **SSLContext.set_default_verify_paths**(). In the future the method may load CA certificates from other locations, too.

The *purpose* flag specifies what kind of CA certificates are loaded. The default settings **Purpose.SERVER_AUTH** loads certificates, that are flagged and trusted for TLS web server
authentication (client side sockets). *Purpose.CLIENT_AUTH* loads CA certificates for client certificate verification on the server side.

```
SSLContext.load_verify_locations(cafile=None, capath=None, cadata=None)
```

Load a set of “certification authority” (CA) certificates used to validate other peers’ certificates when *verify_mode* is other than *CERT_NONE*. At least one of *cafile* or *capath* must be specified.

This method can also load certification revocation lists (CRLs) in PEM or DER format. In order to make use of CRLs, *SSLContext.verify_flags* must be configured properly.

The *cafile* string, if present, is the path to a file of concatenated CA certificates in PEM format. See the discussion of *Certificates* for more information about how to arrange the certificates in this file.

The *capath* string, if present, is the path to a directory containing several CA certificates in PEM format, following an *OpenSSL* specific layout.

The *cadata* object, if present, is either an ASCII string of one or more PEM-encoded certificates or a bytes-like object of DER-encoded certificates. Like with *capath* extra lines around PEM-encoded certificates are ignored but at least one certificate must be present.

```
SSLContext.get_ca_certs(binary_form=False)
```

Get a list of loaded “certification authority” (CA) certificates. If the *binary_form* parameter is *False* each list entry is a dict like the output of *SSLSocket.getpeercert()*. Otherwise the method returns a list of DER-encoded certificates. The returned list does not contain certificates from *capath* unless a certificate was requested and loaded by a SSL connection.
SSLContext.set_default_verify_paths()

Load a set of default “certification authority” (CA) certificates from a filesystem path defined when building the OpenSSL library. Unfortunately, there’s no easy way to know whether this method succeeds: no error is returned if no certificates are to be found. When the OpenSSL library is provided as part of the operating system, though, it is likely to be configured properly.

SSLContext.set_ciphers(ciphers)

Set the available ciphers for sockets created with this context. It should be a string in the OpenSSL cipher list format. If no cipher can be selected (because compile-time options or other configuration forbids use of all the specified ciphers), an SSLError will be raised.

Note: when connected, the SSLSocket.cipher() method of SSL sockets will give the currently selected cipher.

SSLContext.set_alpn_protocols(protocols)

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of ASCII strings, like ['http/1.1', 'spdy/2'], ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to RFC 7301. After a successful handshake, the SSLSocket.selected_alpn_protocol() method will return the agreed-upon protocol.

This method will raise NotImplementedException if HAS_ALPN is False.

OpenSSL 1.1.0+ will abort the handshake and raise SSLError when both sides support ALPN but cannot agree on a protocol.

New in version 2.7.10.
SSLContext.set_npn_protocols(protocols)

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of strings, like ['http/1.1', 'spdy/2'], ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to the NPN draft specification. After a successful handshake, the SSLSocket.selected_npn_protocol() method will return the agreed-upon protocol.

This method will raise NotImplementedException if HAS_NPN is False.

SSLContext.set_servername_callback(server_name_callback)

Register a callback function that will be called after the TLS Client Hello handshake message has been received by the SSL/TLS server when the TLS client specifies a server name indication. The server name indication mechanism is specified in RFC 6066 section 3 - Server Name Indication.

Only one callback can be set per SSLContext. If server_name_callback is None then the callback is disabled. Calling this function a subsequent time will disable the previously registered callback.

The callback function, server_name_callback, will be called with three arguments; the first being the ssl.SSLSocket, the second is a string that represents the server name that the client is intending to communicate (or None if the TLS Client Hello does not contain a server name) and the third argument is the original SSLContext. The server name argument is the IDNA decoded server name.

A typical use of this callback is to change the ssl.SSLSocket's SSLSocket.context attribute to a new object of type SSLContext representing a certificate chain that matches the server name.
Due to the early negotiation phase of the TLS connection, only limited methods and attributes are usable like `SSLSocket.selected_alpn_protocol()` and `SSLSocket.context(SSLSocket.getpeercert(), SSLSocket.getpeercert(), SSLSocket.cipher())` and `SSLSocket.compress()` methods require that the TLS connection has progressed beyond the TLS Client Hello and therefore will not contain return meaningful values nor can they be called safely.

The `server_name_callback` function must return `None` to allow the TLS negotiation to continue. If a TLS failure is required, a constant `ALERT_DESCRIPTION_` can be returned. Other return values will result in a TLS fatal error with `ALERT_DESCRIPTION_INTERNAL_ERROR`.

If there is an IDNA decoding error on the server name, the TLS connection will terminate with an `ALERT_DESCRIPTION_INTERNAL_ERROR` fatal TLS alert message to the client.

If an exception is raised from the `server_name_callback` function the TLS connection will terminate with a fatal TLS alert message `ALERT_DESCRIPTION_HANDSHAKE_FAILURE`.

This method will raise `NotImplementedError` if the OpenSSL library had `OPENSSL_NO_TLSEXT` defined when it was built.

`SSLContext.load_dh_params(dhfile)`

Load the key generation parameters for Diffie-Helman (DH) key exchange. Using DH key exchange improves forward secrecy at the expense of computational resources (both on the server and on the client). The `dhfile` parameter should be the path to a file containing DH parameters in PEM format.
This setting doesn’t apply to client sockets. You can also use the `OP_SINGLE_DH_USE` option to further improve security.

```python
SSLContext.set_ecdh_curve(curve_name)
```

Set the curve name for Elliptic Curve-based Diffie-Hellman (ECDH) key exchange. ECDH is significantly faster than regular DH while arguably as secure. The `curve_name` parameter should be a string describing a well-known elliptic curve, for example `prime256v1` for a widely supported curve.

This setting doesn’t apply to client sockets. You can also use the `OP_SINGLE_ECDH_USE` option to further improve security.

This method is not available if `HAS_ECDH` is `False`.

**See also:**

SSL/TLS & Perfect Forward Secrecy

Vincent Bernat.

```python
SSLContext.wrap_socket(sock, server_side=False, do_handshake_on_connect=True, suppress_ragged_eofs=True, server_hostname=None)
```

Wrap an existing Python socket `sock` and return an `SSLSocket` object. `sock` must be a `SOCK_STREAM` socket; other socket types are unsupported.

The returned SSL socket is tied to the context, its settings and certificates. The parameters `server_side`, `do_handshake_on_connect` and `suppress_ragged_eofs` have the same meaning as in the top-level `wrap_socket()` function.

On client connections, the optional parameter `server_hostname` specifies the hostname of the service which we are connecting to. This allows a single server to host multiple SSL-based
services with distinct certificates, quite similarly to HTTP virtual hosts. Specifying server_hostname will raise a ValueError if server_side is true.

*Changed in version 2.7.9:* Always allow a server_hostname to be passed, even if OpenSSL does not have SNI.

SSLContext.**session_stats**()
Get statistics about the SSL sessions created or managed by this context. A dictionary is returned which maps the names of each piece of information to their numeric values. For example, here is the total number of hits and misses in the session cache since the context was created:

```python
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

SSLContext.**get_ca_certs**(binary_form=False)
Returns a list of dicts with information of loaded CA certs. If the optional argument is true, returns a DER-encoded copy of the CA certificate.

*Note:* Certificates in a capath directory aren't loaded unless they have been used at least once.

SSLContext.**check_hostname**
Wether to match the peer cert's hostname with match_hostname() in SSLSocket.do_handshake(). The context's verify_mode must be set to CERT_OPTIONAL or CERT_REQUIRED, and you must pass server_hostname to wrap_socket() in order to match the hostname.

Example:
```python
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLSv1)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

**Note:** This features requires OpenSSL 0.9.8f or newer.

**SSLContext.options**

An integer representing the set of SSL options enabled on this context. The default value is **OP_ALL**, but you can specify other options such as **OP_NO_SSLv2** by ORing them together.

**Note:** With versions of OpenSSL older than 0.9.8m, it is only possible to set options, not to clear them. Attempting to clear an option (by resetting the corresponding bits) will raise a ValueError.

**SSLContext.protocol**

The protocol version chosen when constructing the context. This attribute is read-only.

**SSLContext.verify_flags**

The flags for certificate verification operations. You can set flags like **VERIFY_CRL_CHECK_LEAF** by ORing them together. By default OpenSSL does neither require nor verify certificate revocation lists (CRLs). Available only with openssl version 0.9.8+.

**SSLContext.verify_mode**

Whether to try to verify other peers' certificates and how to behave if verification fails. This attribute must be one of
CERT_NONE, CERT_OPTIONAL OR CERT_REQUIRED.
17.3.4. Certificates

Certificates in general are part of a public-key / private-key system. In this system, each principal, (which may be a machine, or a person, or an organization) is assigned a unique two-part encryption key. One part of the key is public, and is called the public key; the other part is kept secret, and is called the private key. The two parts are related, in that if you encrypt a message with one of the parts, you can decrypt it with the other part, and only with the other part.

A certificate contains information about two principals. It contains the name of a subject, and the subject’s public key. It also contains a statement by a second principal, the issuer, that the subject is who he claims to be, and that this is indeed the subject’s public key. The issuer’s statement is signed with the issuer’s private key, which only the issuer knows. However, anyone can verify the issuer’s statement by finding the issuer’s public key, decrypting the statement with it, and comparing it to the other information in the certificate. The certificate also contains information about the time period over which it is valid. This is expressed as two fields, called “notBefore” and “notAfter”.

In the Python use of certificates, a client or server can use a certificate to prove who they are. The other side of a network connection can also be required to produce a certificate, and that certificate can be validated to the satisfaction of the client or server that requires such validation. The connection attempt can be set to raise an exception if the validation fails. Validation is done automatically, by the underlying OpenSSL framework; the application need not concern itself with its mechanics. But the application does usually need to provide sets of certificates to allow this process to take place.

Python uses files to contain certificates. They should be formatted as
“PEM” (see RFC 1422), which is a base-64 encoded form wrapped with a header line and a footer line:

```plaintext
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

### 17.3.4.1. Certificate chains

The Python files which contain certificates can contain a sequence of certificates, sometimes called a certificate chain. This chain should start with the specific certificate for the principal who “is” the client or server, and then the certificate for the issuer of that certificate, and then the certificate for the issuer of that certificate, and so on up the chain till you get to a certificate which is self-signed, that is, a certificate which has the same subject and issuer, sometimes called a root certificate. The certificates should just be concatenated together in the certificate file. For example, suppose we had a three certificate chain, from our server certificate to the certificate of the certification authority that signed our server certificate, to the root certificate of the agency which issued the certification authority's certificate:

```plaintext
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

### 17.3.4.2. CA certificates

If you are going to require validation of the other side of the connection's certificate, you need to provide a “CA certs” file, filled
with the certificate chains for each issuer you are willing to trust. Again, this file just contains these chains concatenated together. For validation, Python will use the first chain it finds in the file which matches. The platform's certificates file can be used by calling `SSLContext.load_default_certs()`, this is done automatically with `create_default_context()`.

### 17.3.4.3. Combined key and certificate

Often the private key is stored in the same file as the certificate; in this case, only the `certfile` parameter to `SSLContext.load_cert_chain()` and `wrap_socket()` needs to be passed. If the private key is stored with the certificate, it should come before the first certificate in the certificate chain:

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

### 17.3.4.4. Self-signed certificates

If you are going to create a server that provides SSL-encrypted connection services, you will need to acquire a certificate for that service. There are many ways of acquiring appropriate certificates, such as buying one from a certification authority. Another common practice is to generate a self-signed certificate. The simplest way to do this is with the OpenSSL package, using something like the following:

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout
Generating a 1024 bit RSA private key
.........++++++
..........................++++++
```
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be included into your certificate request.
What you are about to enter is what is called a Distinguished Name (DN). There are quite a few fields but you can leave some blank. For some fields there will be a default value, if you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]: US
State or Province Name (full name) [Some-State]: MyState
Locality Name (e.g., city) []: Some City
Organization Name (e.g., company) [Internet Widgits Pty Ltd]: My Organization
Organizational Unit Name (e.g., section) []: My Group
Common Name (e.g., YOUR name) []: myserver.mygroup.myorganization.com
Email Address []: ops@myserver.mygroup.myorganization.com
%

The disadvantage of a self-signed certificate is that it is its own root certificate, and no one else will have it in their cache of known (and trusted) root certificates.
17.3.5. Examples

17.3.5.1. Testing for SSL support

To test for the presence of SSL support in a Python installation, user code should use the following idiom:

```python
try:
    import ssl
except ImportError:
    pass
else:
    ...  # do something that requires SSL support
```

17.3.5.2. Client-side operation

This example creates a SSL context with the recommended security settings for client sockets, including automatic certificate verification:

```python
>>> context = ssl.create_default_context()
```

If you prefer to tune security settings yourself, you might create a context from scratch (but beware that you might not get the settings right):

```python
>>> context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> context.verify_mode = ssl.CERT_REQUIRED
>>> context.check_hostname = True
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(this snippet assumes your operating system places a bundle of all CA certificates in /etc/ssl/certs/ca-bundle.crt; if not, you’ll get an error and have to adjust the location)

When you use the context to connect to a server, CERT_REQUIRED
validates the server certificate: it ensures that the server certificate was signed with one of the CA certificates, and checks the signature for correctness:

```python
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
   server_hostname="www.python.org")
>>> conn.connect(('www.python.org', 443))
```

You may then fetch the certificate:

```python
>>> cert = conn.getpeercert()
```

Visual inspection shows that the certificate does identify the desired service (that is, the HTTPS host `www.python.org`):

```python
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com/',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationCA.crt',),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
 'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': (((('countryName', 'US'),),
 (((('organizationName', 'DigiCert Inc'),),
 (((('organizationalUnitName', 'www.digicert.com'),),
 (((('commonName', 'DigiCert SHA2 Extended Validation Server CA'),)),
 'notAfter': 'Sep 9 12:00:00 2016 GMT',
 'notBefore': 'Sep 5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': (((('businessCategory', 'Private Organization'),),
 (((('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
 (((('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
 (((('serialNumber', '3359300'),),
 (((('streetAddress', '16 Allen Rd'),),
 (((('postalCode', '03894-4801'),),
 (((('countryName', 'US'),),
 (((('stateOrProvinceName', 'NH'),),
 (((('localityName', 'Wolfeboro'),),
 (((('organizationName', 'Python Software Foundation'),
 (((('commonName', 'www.python.org'),)),
 'subjectAltName': (((('DNS', 'www.python.org'),
 ('DNS', 'python.org'),
 ('DNS', 'pypi.python.org'),
 ('DNS', 'docs.python.org'),
````
Now the SSL channel is established and the certificate verified, you can proceed to talk with the server:

```python
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
 b'Connection: close',
 b'']
```

See the discussion of Security considerations below.

17.3.5.3. Server-side operation

For server operation, typically you’ll need to have a server certificate,
and private key, each in a file. You'll first create a context holding the key and the certificate, so that clients can check your authenticity. Then you'll open a socket, bind it to a port, call `listen()` on it, and start waiting for clients to connect:

```python
import socket, ssl
certfile = "mycertfile"
keyfile = "mykeyfile"

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile=certfile, keyfile=keyfile)

bindsocket = socket.socket()
bindsocket.bind(('myaddr.mydomain.com', 10023))
bindsocket.listen(5)
```

When a client connects, you'll call `accept()` on the socket to get the new socket from the other end, and use the context's `SSLContext.wrap_socket()` method to create a server-side SSL socket for the connection:

```python
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

Then you'll read data from the `connstream` and do something with it till you are finished with the client (or the client is finished with you):

```python
def deal_with_client(connstream):
    data = connstream.read()
    # null data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
```
data = connstream.read()
# finished with client

And go back to listening for new client connections (of course, a real server would probably handle each client connection in a separate thread, or put the sockets in non-blocking mode and use an event loop).
When working with non-blocking sockets, there are several things you need to be aware of:

- Calling `select()` tells you that the OS-level socket can be read from (or written to), but it does not imply that there is sufficient data at the upper SSL layer. For example, only part of an SSL frame might have arrived. Therefore, you must be ready to handle `SSLSocket.recv()` and `SSLSocket.send()` failures, and retry after another call to `select()`.

- Conversely, since the SSL layer has its own framing, a SSL socket may still have data available for reading without `select()` being aware of it. Therefore, you should first call `SSLSocket.recv()` to drain any potentially available data, and then only block on a `select()` call if still necessary.

(of course, similar provisions apply when using other primitives such as `poll()`, or those in the `selectors` module)

- The SSL handshake itself will be non-blocking: the `SSLSocket.do_handshake()` method has to be retried until it returns successfully. Here is a synopsis using `select()` to wait for the socket's readiness:

```python
while True:
    try:
        sock.do_handshake()
        break
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```
17.3.7. Security considerations

17.3.7.1. Best defaults

For client use, if you don’t have any special requirements for your security policy, it is highly recommended that you use the create_default_context() function to create your SSL context. It will load the system’s trusted CA certificates, enable certificate validation and hostname checking, and try to choose reasonably secure protocol and cipher settings.

If a client certificate is needed for the connection, it can be added with SSLContext.load_cert_chain().

By contrast, if you create the SSL context by calling the SSLContext constructor yourself, it will not have certificate validation nor hostname checking enabled by default. If you do so, please read the paragraphs below to achieve a good security level.

17.3.7.2. Manual settings

17.3.7.2.1. Verifying certificates

When calling the SSLContext constructor directly, CERT_NONE is the default. Since it does not authenticate the other peer, it can be insecure, especially in client mode where most of time you would like to ensure the authenticity of the server you’re talking to. Therefore, when in client mode, it is highly recommended to use CERT_REQUIRED. However, it is in itself not sufficient; you also have to check that the server certificate, which can be obtained by calling SSLSocket.getpeercert(), matches the desired service. For many protocols and applications, the service can be identified by the
hostname; in this case, the `match_hostname()` function can be used. This common check is automatically performed when `SSLContext.check_hostname` is enabled.

In server mode, if you want to authenticate your clients using the SSL layer (rather than using a higher-level authentication mechanism), you'll also have to specify `CERT_REQUIRED` and similarly check the client certificate.

**Note:** In client mode, `CERT_OPTIONAL` and `CERT_REQUIRED` are equivalent unless anonymous ciphers are enabled (they are disabled by default).

17.3.7.2.2. Protocol versions

SSL versions 2 and 3 are considered insecure and are therefore dangerous to use. If you want maximum compatibility between clients and servers, it is recommended to use `PROTOCOL_SSLv23` as the protocol version and then disable SSLv2 and SSLv3 explicitly using the `SSLContext.options` attribute:

```python
context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
context.options |= ssl.OP_NO_SSLv2
context.options |= ssl.OP_NO_SSLv3
```

The SSL context created above will only allow TLSv1 and later (if supported by your system) connections.

17.3.7.2.3. Cipher selection

If you have advanced security requirements, fine-tuning of the ciphers enabled when negotiating a SSL session is possible through the `SSLContext.set_ciphers()` method. Starting from Python 2.7.9, the ssl module disables certain weak ciphers by default, but you may
want to further restrict the cipher choice. Be sure to read OpenSSL’s documentation about the cipher list format. If you want to check which ciphers are enabled by a given cipher list, use the `openssl ciphers` command on your system.

17.3.7.3. Multi-processing

If using this module as part of a multi-processed application (using, for example the `multiprocessing` or `concurrent.futures` modules), be aware that OpenSSL’s internal random number generator does not properly handle forked processes. Applications must change the PRNG state of the parent process if they use any SSL feature with `os.fork()`. Any successful call of `RAND_add()`, `RAND_bytes()` or `RAND_pseudo_bytes()` is sufficient.

See also:

Class `socket.socket`

Documentation of underlying `socket` class

SSL/TLS Strong Encryption: An Introduction

Intro from the Apache webserver documentation

RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management

Steve Kent

RFC 1750: Randomness Recommendations for Security

D. Eastlake et. al.

RFC 3280: Internet X.509 Public Key Infrastructure Certificate and CRL Profile

Housley et. al.

RFC 4366: Transport Layer Security (TLS) Extensions

Blake-Wilson et. al.

Version 1.2
T. Dierks et. al.

RFC 6066: Transport Layer Security (TLS) Extensions
D. Eastlake

IANA TLS: Transport Layer Security (TLS) Parameters
IANA
17.4. `signal` — Set handlers for asynchronous events

This module provides mechanisms to use signal handlers in Python. Some general rules for working with signals and their handlers:

- A handler for a particular signal, once set, remains installed until it is explicitly reset (Python emulates the BSD style interface regardless of the underlying implementation), with the exception of the handler for `SIGCHLD`, which follows the underlying implementation.
- There is no way to “block” signals temporarily from critical sections (since this is not supported by all Unix flavors).
- Although Python signal handlers are called asynchronously as far as the Python user is concerned, they can only occur between the “atomic” instructions of the Python interpreter. This means that signals arriving during long calculations implemented purely in C (such as regular expression matches on large bodies of text) may be delayed for an arbitrary amount of time.
- When a signal arrives during an I/O operation, it is possible that the I/O operation raises an exception after the signal handler returns. This is dependent on the underlying Unix system’s semantics regarding interrupted system calls.
- Because the C signal handler always returns, it makes little sense to catch synchronous errors like `SIGFPE` or `SIGSEGV`.
- Python installs a small number of signal handlers by default: `SIGPIPE` is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions) and `SIGINT` is translated into a `KeyboardInterrupt` exception. All of these can be overridden.
- Some care must be taken if both signals and threads are used
in the same program. The fundamental thing to remember in using signals and threads simultaneously is: always perform `signal()` operations in the main thread of execution. Any thread can perform an `alarm()`, `getsignal()`, `pause()`, `setitimer()` or `getitimer()`; only the main thread can set a new signal handler, and the main thread will be the only one to receive signals (this is enforced by the Python `signal` module, even if the underlying thread implementation supports sending signals to individual threads). This means that signals can't be used as a means of inter-thread communication. Use locks instead.

The variables defined in the `signal` module are:

`signal.SIG_DFL`
This is one of two standard signal handling options; it will simply perform the default function for the signal. For example, on most systems the default action for `SIGQUIT` is to dump core and exit, while the default action for `SIGCHLD` is to simply ignore it.

`signal.SIG_IGN`
This is another standard signal handler, which will simply ignore the given signal.

`SIG*`
All the signal numbers are defined symbolically. For example, the hangup signal is defined as `signal.SIGHUP`; the variable names are identical to the names used in C programs, as found in `<signal.h>`. The Unix man page for `signal()` lists the existing signals (on some systems this is `signal(2)`, on others the list is in `signal(7)`). Note that not all systems define the same set of signal names; only those names defined by the system are defined by this module.

`signal.CTRL_C_EVENT`
The signal corresponding to the \texttt{Ctrl+C} keystroke event. This signal can only be used with \texttt{os.kill()}.

Availability: Windows.

\textit{New in version 2.7.}

\texttt{signal.\texttt{CTRL\_BREAK\_EVENT}}

The signal corresponding to the \texttt{Ctrl+Break} keystroke event. This signal can only be used with \texttt{os.kill()}.

Availability: Windows.

\textit{New in version 2.7.}

\texttt{signal.\texttt{NSIG}}

One more than the number of the highest signal number.

\texttt{signal.\texttt{ITIMER\_REAL}}

Decrements interval timer in real time, and delivers \texttt{SIGALRM} upon expiration.

\texttt{signal.\texttt{ITIMER\_VIRTUAL}}

Decrements interval timer only when the process is executing, and delivers \texttt{SIGVTALRM} upon expiration.

\texttt{signal.\texttt{ITIMER\_PROF}}

Decrements interval timer both when the process executes and when the system is executing on behalf of the process. Coupled with \texttt{ITIMER\_VIRTUAL}, this timer is usually used to profile the time spent by the application in user and kernel space. \texttt{SIGPROF} is delivered upon expiration.

The \texttt{signal} module defines one exception:

\textit{exception signal.\texttt{ItimerError}}
Raised to signal an error from the underlying `setitimer()` or `getitimer()` implementation. Expect this error if an invalid interval timer or a negative time is passed to `setitimer()`. This error is a subtype of `IOError`.

The `signal` module defines the following functions:

**signal.** `alarm(time)`

If `time` is non-zero, this function requests that a `SIGALRM` signal be sent to the process in `time` seconds. Any previously scheduled alarm is canceled (only one alarm can be scheduled at any time). The returned value is then the number of seconds before any previously set alarm was to have been delivered. If `time` is zero, no alarm is scheduled, and any scheduled alarm is canceled. If the return value is zero, no alarm is currently scheduled. (See the Unix man page `alarm(2)`) Availability: Unix.

**signal.** `getsignal(signalnum)`

Return the current signal handler for the signal `signalnum`. The returned value may be a callable Python object, or one of the special values `signal.SIG_IGN`, `signal.SIG_DFL` or `None`. Here, `signal.SIG_IGN` means that the signal was previously ignored, `signal.SIG_DFL` means that the default way of handling the signal was previously in use, and `None` means that the previous signal handler was not installed from Python.

**signal.** `pause()`

Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing. Not on Windows. (See the Unix man page `signal(2)`.)

**signal.** `setitimer(which, seconds[, interval])`

Sets given interval timer (one of `signal.ITIMER_REAL`,
`signal.ITIMER_VIRTUAL` or `signal.ITIMER_PROF`) specified by `which` to fire after `seconds` (float is accepted, different from `alarm()`) and after that every `interval` seconds. The interval timer specified by `which` can be cleared by setting `seconds` to zero.

When an interval timer fires, a signal is sent to the process. The signal sent is dependent on the timer being used; `signal.ITIMER_REAL` will deliver `SIGALRM`, `signal.ITIMER_VIRTUAL` sends `SIGVTALRM`, and `signal.ITIMER_PROF` will deliver `SIGPROF`.

The old values are returned as a tuple: `(delay, interval)`.

Attempting to pass an invalid interval timer will cause an `ItimerError`. Availability: Unix.

*New in version 2.6.*

`signal.getitimer(`which`)`

Returns current value of a given interval timer specified by `which`. Availability: Unix.

*New in version 2.6.*

`signal.set_wakeup_fd(`fd`)`

Set the wakeup fd to `fd`. When a signal is received, a `
\0` byte is written to the fd. This can be used by a library to wakeup a poll or select call, allowing the signal to be fully processed.

The old wakeup fd is returned. `fd` must be non-blocking. It is up to the library to remove any bytes before calling poll or select again.

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a `ValueError` exception to be raised.

*New in version 2.6.*
**signal.** `siginterrupt(signalnum, flag)`

Change system call restart behaviour: if flag is `False`, system calls will be restarted when interrupted by signal `signalnum`, otherwise system calls will be interrupted. Returns nothing. Availability: Unix (see the man page `siginterrupt(3)` for further information).

Note that installing a signal handler with `signal()` will reset the restart behaviour to interruptible by implicitly calling `siginterrupt()` with a true `flag` value for the given signal.

*New in version 2.6.*

**signal.** `signal(signalnum, handler)`

Set the handler for signal `signalnum` to the function `handler`. `handler` can be a callable Python object taking two arguments (see below), or one of the special values `signal.SIG_IGN` or `signal.SIG_DFL`. The previous signal handler will be returned (see the description of `getsignal()` above). (See the Unix man page `signal(2)`.)

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a `ValueError` exception to be raised.

The `handler` is called with two arguments: the signal number and the current stack frame (`None` or a frame object; for a description of frame objects, see the description in the type hierarchy or see the attribute descriptions in the `inspect` module).

On Windows, `signal()` can only be called with `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, or `SIGTERM`. A `ValueError` will be raised in any other case.
17.4.1. Example

Here is a minimal example program. It uses the `alarm()` function to limit the time spent waiting to open a file; this is useful if the file is for a serial device that may not be turned on, which would normally cause the `os.open()` to hang indefinitely. The solution is to set a 5-second alarm before opening the file; if the operation takes too long, the alarm signal will be sent, and the handler raises an exception.

```python
import signal, os

def handler(signum, frame):
    print 'Signal handler called with signal', signum
    raise IOError("Couldn't open device!")

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)    # Disable the alarm
```

?
17.5. `popen2` — Subprocesses with accessible I/O streams

 Deprecated since version 2.6: This module is obsolete. Use the `subprocess` module. Check especially the Replacing Older Functions with the subprocess Module section.

This module allows you to spawn processes and connect to their input/output/error pipes and obtain their return codes under Unix and Windows.

The `subprocess` module provides more powerful facilities for spawning new processes and retrieving their results. Using the `subprocess` module is preferable to using the `popen2` module.

The primary interface offered by this module is a trio of factory functions. For each of these, if `bufsize` is specified, it specifies the buffer size for the I/O pipes. `mode`, if provided, should be the string 'b' or 't'; on Windows this is needed to determine whether the file objects should be opened in binary or text mode. The default value for `mode` is 't'.

On Unix, `cmd` may be a sequence, in which case arguments will be passed directly to the program without shell intervention (as with `os.spawnv()`). If `cmd` is a string it will be passed to the shell (as with `os.system()`).

The only way to retrieve the return codes for the child processes is by using the `poll()` or `wait()` methods on the `Popen3` and `Popen4` classes; these are only available on Unix. This information is not available when using the `popen2()`, `popen3()`, and `popen4()`
functions, or the equivalent functions in the `os` module. (Note that the
tuples returned by the `os` module's functions are in a different order
from the ones returned by the `popen2` module.)

```
popen2. `popen2(cmd[, bufsize[, mode]])`
  Executes `cmd` as a sub-process. Returns the file objects
  (child_stdout, child_stdin).
```

```
popen2. `popen3(cmd[, bufsize[, mode]])`
  Executes `cmd` as a sub-process. Returns the file objects
  (child_stdout, child_stdin, child_stderr).
```

```
popen2. `popen4(cmd[, bufsize[, mode]])`
  Executes `cmd` as a sub-process. Returns the file objects
  (child_stdout_and_stderr, child_stdin).
```

*New in version 2.0.*

On Unix, a class defining the objects returned by the factory
functions is also available. These are not used for the Windows
implementation, and are not available on that platform.

```
class popen2. `Popen3(cmd[, capturestderr[, bufsize]])`
  This class represents a child process. Normally, `Popen3` instances
  are created using the `popen2()` and `popen3()` factory functions
described above.
```

If not using one of the helper functions to create `Popen3` objects,
the parameter `cmd` is the shell command to execute in a sub-
process. The `capturestderr` flag, if true, specifies that the object
should capture standard error output of the child process. The
default is false. If the `bufsize` parameter is specified, it specifies
the size of the I/O buffers to/from the child process.
class popen2. **Popen4**(cmd[, bufsize])

Similar to **Popen3**, but always captures standard error into the same file object as standard output. These are typically created using **popen4()**.

*New in version 2.0.*
17.5.1. Popen3 and Popen4 Objects

Instances of the `Popen3` and `Popen4` classes have the following methods:

**Popen3. poll()**

Returns `-1` if child process hasn’t completed yet, or its status code (see `wait()`) otherwise.

**Popen3. wait()**

Waits for and returns the status code of the child process. The status code encodes both the return code of the process and information about whether it exited using the `exit()` system call or died due to a signal. Functions to help interpret the status code are defined in the `os` module; see section Process Management for the `W*()` family of functions.

The following attributes are also available:

**Popen3. fromchild**

A file object that provides output from the child process. For `Popen4` instances, this will provide both the standard output and standard error streams.

**Popen3. tochild**

A file object that provides input to the child process.

**Popen3. childerr**

A file object that provides error output from the child process, if `capturestderr` was true for the constructor, otherwise `None`. This will always be `None` for `Popen4` instances.

**Popen3. pid**

The process ID of the child process.
17.5.2. Flow Control Issues

Any time you are working with any form of inter-process communication, control flow needs to be carefully thought out. This remains the case with the file objects provided by this module (or the `os` module equivalents).

When reading output from a child process that writes a lot of data to standard error while the parent is reading from the child's standard output, a deadlock can occur. A similar situation can occur with other combinations of reads and writes. The essential factors are that more than `_PC_PIPE_BUF` bytes are being written by one process in a blocking fashion, while the other process is reading from the first process, also in a blocking fashion.

There are several ways to deal with this situation.

The simplest application change, in many cases, will be to follow this model in the parent process:

```python
import popen2
r, w, e = popen2.popen3('python slave.py')
e.readlines()
r.readlines()
r.close()
e.close()
w.close()
```

with code like this in the child:

```python
import os
import sys

# note that each of these print statements
# writes a single long string
```
print >>sys.stderr, 400 * 'this is a test\n'
os.close(sys.stderr.fileno())
print >>sys.stdout, 400 * 'this is another test\n'

In particular, note that `sys.stderr` must be closed after writing all data, or `readlines()` won’t return. Also note that `os.close()` must be used, as `sys.stderr.close()` won’t close `stderr` (otherwise assigning to `sys.stderr` will silently close it, so no further errors can be printed).

Applications which need to support a more general approach should integrate I/O over pipes with their `select()` loops, or use separate threads to read each of the individual files provided by whichever `popen*()` function or `Popen*` class was used.

**See also:**

**Module** `subprocess`
Module for spawning and managing subprocesses.
This module provides the basic infrastructure for writing asynchronous socket service clients and servers.

There are only two ways to have a program on a single processor do “more than one thing at a time.” Multi-threaded programming is the simplest and most popular way to do it, but there is another very different technique, that lets you have nearly all the advantages of multi-threading, without actually using multiple threads. It's really only practical if your program is largely I/O bound. If your program is processor bound, then pre-emptive scheduled threads are probably what you really need. Network servers are rarely processor bound, however.

If your operating system supports the `select()` system call in its I/O library (and nearly all do), then you can use it to juggle multiple communication channels at once; doing other work while your I/O is taking place in the “background.” Although this strategy can seem strange and complex, especially at first, it is in many ways easier to understand and control than multi-threaded programming. The `asyncore` module solves many of the difficult problems for you, making the task of building sophisticated high-performance network servers and clients a snap. For “conversational” applications and protocols the companion `asynchat` module is invaluable.

The basic idea behind both modules is to create one or more network `channels`, instances of class `asyncore.dispatcher` and `asynchat.async_chat`. Creating the channels adds them to a global
map, used by the `loop()` function if you do not provide it with your own `map`.

Once the initial channel(s) is(are) created, calling the `loop()` function activates channel service, which continues until the last channel (including any that have been added to the map during asynchronous service) is closed.

```
asyncore.loop([timeout[, use_poll[, map[, count]]]])
```

Enter a polling loop that terminates after count passes or all open channels have been closed. All arguments are optional. The `count` parameter defaults to `None`, resulting in the loop terminating only when all channels have been closed. The `timeout` argument sets the timeout parameter for the appropriate `select()` or `poll()` call, measured in seconds; the default is 30 seconds. The `use_poll` parameter, if true, indicates that `poll()` should be used in preference to `select()` (the default is `False`).

The `map` parameter is a dictionary whose items are the channels to watch. As channels are closed they are deleted from their map. If `map` is omitted, a global map is used. Channels (instances of `asyncore.dispatcher`, `asynchat.async_chat` and subclasses thereof) can freely be mixed in the map.

```
class asyncore.dispatcher
```

The `dispatcher` class is a thin wrapper around a low-level socket object. To make it more useful, it has a few methods for event-handling which are called from the asynchronous loop. Otherwise, it can be treated as a normal non-blocking socket object.

The firing of low-level events at certain times or in certain connection states tells the asynchronous loop that certain higher-level events have taken place. For example, if we have asked for
a socket to connect to another host, we know that the connection has been made when the socket becomes writable for the first time (at this point you know that you may write to it with the expectation of success). The implied higher-level events are:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle_connect()</td>
<td>Implied by the first read or write event</td>
</tr>
<tr>
<td>handle_close()</td>
<td>Implied by a read event with no data available</td>
</tr>
<tr>
<td>handle_accept()</td>
<td>Implied by a read event on a listening socket</td>
</tr>
</tbody>
</table>

During asynchronous processing, each mapped channel's `readable()` and `writable()` methods are used to determine whether the channel's socket should be added to the list of channels `select()`ed or `poll()`ed for read and write events.

Thus, the set of channel events is larger than the basic socket events. The full set of methods that can be overridden in your subclass follows:

**handle_read()**

Called when the asynchronous loop detects that a `read()` call on the channel's socket will succeed.

**handle_write()**

Called when the asynchronous loop detects that a writable socket can be written. Often this method will implement the necessary buffering for performance. For example:

```python
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

**handle_expt()**

Called when there is out of band (OOB) data for a socket
connection. This will almost never happen, as OOB is tenuously supported and rarely used.

**handle_connect()**
Called when the active opener’s socket actually makes a connection. Might send a “welcome” banner, or initiate a protocol negotiation with the remote endpoint, for example.

**handle_close()**
Called when the socket is closed.

**handle_error()**
Called when an exception is raised and not otherwise handled. The default version prints a condensed traceback.

**handle_accept()**
Called on listening channels (passive openers) when a connection can be established with a new remote endpoint that has issued a `connect()` call for the local endpoint.

**readable()**
Called each time around the asynchronous loop to determine whether a channel’s socket should be added to the list on which read events can occur. The default method simply returns `True`, indicating that by default, all channels will be interested in read events.

**writable()**
Called each time around the asynchronous loop to determine whether a channel's socket should be added to the list on which write events can occur. The default method simply returns `True`, indicating that by default, all channels will be interested in write events.

In addition, each channel delegates or extends many of the
socket methods. Most of these are nearly identical to their socket partners.

**create_socket**(family, type)

This is identical to the creation of a normal socket, and will use the same options for creation. Refer to the socket documentation for information on creating sockets.

**connect**(address)

As with the normal socket object, address is a tuple with the first element the host to connect to, and the second the port number.

**send**(data)

Send data to the remote end-point of the socket.

**recv**(buffer_size)

Read at most buffer_size bytes from the socket’s remote end-point. An empty string implies that the channel has been closed from the other end.

Note that recv() may raise socket.error with EAGAIN or EWOULDBLOCK, even though select.select() or select.poll() has reported the socket ready for reading.

**listen**(backlog)

Listen for connections made to the socket. The backlog argument specifies the maximum number of queued connections and should be at least 1; the maximum value is system-dependent (usually 5).

**bind**(address)

Bind the socket to address. The socket must not already be bound. (The format of address depends on the address family — refer to the socket documentation for more information.) To
mark the socket as re-usable (setting the SO_REUSEADDR option), call the dispatcher object's set_reuse_addr() method.

```
accept()
```

Accept a connection. The socket must be bound to an address and listening for connections. The return value can be either None or a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection. When None is returned it means the connection didn’t take place, in which case the server should just ignore this event and keep listening for further incoming connections.

```
close()
```

Close the socket. All future operations on the socket object will fail. The remote end-point will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

```
class asyncore.dispatcher_with_send
```

A dispatcher subclass which adds simple buffered output capability, useful for simple clients. For more sophisticated usage use asyncchat.async_chat.

```
class asyncore.file_dispatcher
```

A file_dispatcher takes a file descriptor or file object along with an optional map argument and wraps it for use with the poll() or loop() functions. If provided a file object or anything with a fileno() method, that method will be called and passed to the file_wrapper constructor. Availability: UNIX.

```
class asyncore.file_wrapper
```
A file_wrapper takes an integer file descriptor and calls `os.dup()` to duplicate the handle so that the original handle may be closed independently of the file_wrapper. This class implements sufficient methods to emulate a socket for use by the `file_dispatcher` class. Availability: UNIX.
Here is a very basic HTTP client that uses the `dispatcher` class to implement its socket handling:

```python
import asyncore, socket

class HTTPClient(asyncore.dispatcher):
    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.connect( (host, 80) )
        self.buffer = 'GET %s HTTP/1.0\r\n\r\n' % path

    def handle_connect(self):
        pass

    def handle_close(self):
        self.close()

    def handle_read(self):
        print self.recv(8192)

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]

client = HTTPClient('www.python.org', '/')
asyncore.loop()
```
17.6.2. asyncore Example basic echo server

Here is a basic echo server that uses the `dispatcher` class to accept connections and dispatches the incoming connections to a handler:

```python
import asyncore
import socket

class EchoHandler(asyncore.dispatcher_with_send):
    def handle_read(self):
        data = self.recv(8192)
        if data:
            self.send(data)

class EchoServer(asyncore.dispatcher):
    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.set_reuse_addr()
        self.bind(((host, port)))
        self.listen(5)

    def handle_accept(self):
        pair = self.accept()
        if pair is not None:
            sock, addr = pair
            print 'Incoming connection from %s' % repr(addr)
            handler = EchoHandler(sock)

server = EchoServer('localhost', 8080)
asyncore.loop()
```
This module builds on the `asyncore` infrastructure, simplifying asynchronous clients and servers and making it easier to handle protocols whose elements are terminated by arbitrary strings, or are of variable length. `asynchat` defines the abstract class `async_chat` that you subclass, providing implementations of the `collect_incoming_data()` and `found_terminator()` methods. It uses the same asynchronous loop as `asyncore`, and the two types of channel, `asyncore.dispatcher` and `asynchat.async_chat`, can freely be mixed in the channel map. Typically an `asyncore.dispatcher` server channel generates new `asynchat.async_chat` channel objects as it receives incoming connection requests.

```python
class asynchat.async_chat
    This class is an abstract subclass of `asyncore.dispatcher`. To make practical use of the code you must subclass `async_chat`, providing meaningful `collect_incoming_data()` and `found_terminator()` methods. The `asyncore.dispatcher` methods can be used, although not all make sense in a message/response context.

    Like `asyncore.dispatcher`, `async_chat` defines a set of events that are generated by an analysis of socket conditions after a `select()` call. Once the polling loop has been started the `async_chat` object’s methods are called by the event-processing framework with no action on the part of the programmer.
```
Two class attributes can be modified, to improve performance, or possibly even to conserve memory.

**ac_in_buffer_size**
The asynchronous input buffer size (default 4096).

**ac_out_buffer_size**
The asynchronous output buffer size (default 4096).

Unlike `asyncore.dispatcher`, `async_chat` allows you to define a first-in-first-out queue (fifo) of *producers*. A producer need have only one method, `more()`, which should return data to be transmitted on the channel. The producer indicates exhaustion (i.e. that it contains no more data) by having its `more()` method return the empty string. At this point the `async_chat` object removes the producer from the fifo and starts using the next producer, if any. When the producer fifo is empty the `handle_write()` method does nothing. You use the channel object’s `set_terminator()` method to describe how to recognize the end of, or an important breakpoint in, an incoming transmission from the remote endpoint.

To build a functioning `async_chat` subclass your input methods `collect_incoming_data()` and `found_terminator()` must handle the data that the channel receives asynchronously. The methods are described below.

`async_chat.close_when_done()`
Pushes a `None` on to the producer fifo. When this producer is popped off the fifo it causes the channel to be closed.

`async_chat.collect_incoming_data(data)`
Called with `data` holding an arbitrary amount of received data. The default method, which must be overridden, raises a
**NotImplementedError** exception.

**async_chat.discard_buffers()**
In emergencies this method will discard any data held in the input and/or output buffers and the producer fifo.

**async_chat.found_terminator()**
Called when the incoming data stream matches the termination condition set by `set_terminator()`. The default method, which must be overridden, raises a **NotImplementedError** exception. The buffered input data should be available via an instance attribute.

**async_chat.get_terminator()**
Returns the current terminator for the channel.

**async_chat.push(data)**
Pushes data on to the channel's fifo to ensure its transmission. This is all you need to do to have the channel write the data out to the network, although it is possible to use your own producers in more complex schemes to implement encryption and chunking, for example.

**async_chat.push_with_producer(producer)**
Takes a producer object and adds it to the producer fifo associated with the channel. When all currently-pushed producers have been exhausted the channel will consume this producer’s data by calling its `more()` method and send the data to the remote endpoint.

**async_chat.set_terminator(term)**
Sets the terminating condition to be recognized on the channel. **term** may be any of three types of value, corresponding to three different ways to handle incoming protocol data.

<table>
<thead>
<tr>
<th>term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>string</strong></td>
<td>Will call <code>found_terminator()</code> when the string is found in the input stream</td>
</tr>
<tr>
<td><strong>integer</strong></td>
<td>Will call <code>found_terminator()</code> when the indicated number of characters have been received</td>
</tr>
<tr>
<td><strong>None</strong></td>
<td>The channel continues to collect data forever</td>
</tr>
</tbody>
</table>

Note that any data following the terminator will be available for reading by the channel after `found_terminator()` is called.
17.7.1. asynchat - Auxiliary Classes

class asynchat.fifo([list=None])

A fifo holding data which has been pushed by the application but not yet popped for writing to the channel. A fifo is a list used to hold data and/or producers until they are required. If the list argument is provided then it should contain producers or data items to be written to the channel.

is_empty()

Returns True if and only if the fifo is empty.

first()

Returns the least-recently push()ed item from the fifo.

push(data)

Adds the given data (which may be a string or a producer object) to the producer fifo.

pop()

If the fifo is not empty, returns True, first(), deleting the popped item. Returns False, None for an empty fifo.
17.7.2. asynchat Example

The following partial example shows how HTTP requests can be read with `async_chat`. A web server might create an `http_request_handler` object for each incoming client connection. Notice that initially the channel terminator is set to match the blank line at the end of the HTTP headers, and a flag indicates that the headers are being read.

Once the headers have been read, if the request is of type POST (indicating that further data are present in the input stream) then the `Content-Length:` header is used to set a numeric terminator to read the right amount of data from the channel.

The `handle_request()` method is called once all relevant input has been marshalled, after setting the channel terminator to `None` to ensure that any extraneous data sent by the web client are ignored.

```python
class http_request_handler(asynchat.async_chat):
    def __init__(self, sock, addr, sessions, log):
        asynchat.async_chat.__init__(self, sock=sock)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = ""
        self.set_terminator("\r\n\r\n")
        self.reading_headers = True
        self.handling = False
        self.cgi_data = None
        self.log = log

    def collect_incoming_data(self, data):
        """Buffer the data""
        self.ibuffer.append(data)

    def found_terminator(self):
        if self.reading_headers:
```
self.reading_headers = False
self.parse_headers("".join(self.ibuffer))
self.ibuffer = []
if self.op.upper() == "POST":
    clen = self.headers.getheader("content-length")
    self.set_terminator(int(clen))
else:
    self.handling = True
    self.set_terminator(None)
    self.handle_request()
elif not self.handling:
    self.set_terminator(None)  # browsers sometimes ove
    self.cgi_data = parse(self.headers, "".join(self.ib
    self.handling = True
    self.ibuffer = []
    self.handle_request()
18. Internet Data Handling

This chapter describes modules which support handling data formats commonly used on the Internet.

- **18.1. email** — An email and MIME handling package
  - 18.1.1. email.message: Representing an email message
  - 18.1.2. email.parser: Parsing email messages
    - 18.1.2.1. FeedParser API
    - 18.1.2.2. Parser class API
    - 18.1.2.3. Additional notes
  - 18.1.3. email.generator: Generating MIME documents
  - 18.1.4. email.mime: Creating email and MIME objects from scratch
  - 18.1.5. email.header: Internationalized headers
  - 18.1.6. email.charset: Representing character sets
  - 18.1.7. email.encoders: Encoders
  - 18.1.8. email.errors: Exception and Defect classes
  - 18.1.9. email.utils: Miscellaneous utilities
  - 18.1.10. email.iterators: Iterators
  - 18.1.11. email: Examples
  - 18.1.12. Package History
  - 18.1.13. Differences from mimelib

- **18.2. json** — JSON encoder and decoder
  - 18.2.1. Basic Usage
  - 18.2.2. Encoders and Decoders
  - 18.2.3. Standard Compliance and Interoperability
    - 18.2.3.1. Character Encodings
    - 18.2.3.2. Infinite and NaN Number Values
    - 18.2.3.3. Repeated Names Within an Object
    - 18.2.3.4. Top-level Non-Object, Non-Array Values
    - 18.2.3.5. Implementation Limitations
• 18.3. mailcap — Mailcap file handling
• 18.4. mailbox — Manipulate mailboxes in various formats
  ○ 18.4.1. Mailbox objects
    ▪ 18.4.1.1. Maildir
    ▪ 18.4.1.2. mbox
    ▪ 18.4.1.3. MH
    ▪ 18.4.1.4. Babyl
    ▪ 18.4.1.5. MMDF
  ○ 18.4.2. Message objects
    ▪ 18.4.2.1. MaildirMessage
    ▪ 18.4.2.2. mboxMessage
    ▪ 18.4.2.3. MHMessage
    ▪ 18.4.2.4. BabylMessage
    ▪ 18.4.2.5. MMDFMessage
  ○ 18.4.3. Exceptions
  ○ 18.4.4. Deprecated classes and methods
  ○ 18.4.5. Examples
• 18.5. mhlib — Access to MH mailboxes
  ○ 18.5.1. MH Objects
  ○ 18.5.2. Folder Objects
  ○ 18.5.3. Message Objects
• 18.6. mimetools — Tools for parsing MIME messages
  ○ 18.6.1. Additional Methods of Message Objects
• 18.7. mimetypes — Map filenames to MIME types
  ○ 18.7.1. MimeTypes Objects
• 18.8. MimeWriter — Generic MIME file writer
  ○ 18.8.1. MimeWriter Objects
• 18.9. mimify — MIME processing of mail messages
• 18.10. multfile — Support for files containing distinct parts
  ○ 18.10.1. MultiFile Objects
  ○ 18.10.2. MultiFile Example
• 18.11. rfc822 — Parse RFC 2822 mail headers
• 18.11.1. Message Objects
• 18.11.2. AddressList Objects
• 18.12. **base64** — RFC 3548: Base16, Base32, Base64 Data Encodings
• 18.13. **binhex** — Encode and decode binhex4 files
  • 18.13.1. Notes
• 18.14. **binascii** — Convert between binary and ASCII
• 18.15. **quopri** — Encode and decode MIME quoted-printable data
• 18.16. **uu** — Encode and decode uuencode files
18.1. email — An email and MIME handling package

New in version 2.2.

The email package is a library for managing email messages, including MIME and other RFC 2822-based message documents. It subsumes most of the functionality in several older standard modules such as rfc822, mimetools, multifile, and other non-standard packages such as mimecnt1. It is specifically not designed to do any sending of email messages to SMTP (RFC 2821), NNTP, or other servers; those are functions of modules such as smtplib and nntplib. The email package attempts to be as RFC-compliant as possible, supporting in addition to RFC 2822, such MIME-related RFCs as RFC 2045, RFC 2046, RFC 2047, and RFC 2231.

The primary distinguishing feature of the email package is that it splits the parsing and generating of email messages from the internal object model representation of email. Applications using the email package deal primarily with objects; you can add sub-objects to messages, remove sub-objects from messages, completely re-arrange the contents, etc. There is a separate parser and a separate generator which handles the transformation from flat text to the object model, and then back to flat text again. There are also handy subclasses for some common MIME object types, and a few miscellaneous utilities that help with such common tasks as extracting and parsing message field values, creating RFC-compliant dates, etc.

The following sections describe the functionality of the email package. The ordering follows a progression that should be common in applications: an email message is read as flat text from a file or
other source, the text is parsed to produce the object structure of the email message, this structure is manipulated, and finally, the object tree is rendered back into flat text.

It is perfectly feasible to create the object structure out of whole cloth — i.e. completely from scratch. From there, a similar progression can be taken as above.

Also included are detailed specifications of all the classes and modules that the email package provides, the exception classes you might encounter while using the email package, some auxiliary utilities, and a few examples. For users of the older mimelib package, or previous versions of the email package, a section on differences and porting is provided.

Contents of the email package documentation:

- **18.1.1. email.message**: Representing an email message
- **18.1.2. email.parser**: Parsing email messages
  - **18.1.2.1. FeedParser API**
  - **18.1.2.2. Parser class API**
  - **18.1.2.3. Additional notes**
- **18.1.3. email.generator**: Generating MIME documents
- **18.1.4. email.mime**: Creating email and MIME objects from scratch
- **18.1.5. email.header**: Internationalized headers
- **18.1.6. email.charset**: Representing character sets
- **18.1.7. email.encoders**: Encoders
- **18.1.8. email.errors**: Exception and Defect classes
- **18.1.9. email.utils**: Miscellaneous utilities
- **18.1.10. email.iterators**: Iterators
- **18.1.11. email**: Examples
See also:

**Module smtplib**
  SMTP protocol client

**Module nntplib**
  NNTP protocol client
18.1.12. Package History

This table describes the release history of the email package, corresponding to the version of Python that the package was released with. For purposes of this document, when you see a note about change or added versions, these refer to the Python version the change was made in, not the email package version. This table also describes the Python compatibility of each version of the package.

<table>
<thead>
<tr>
<th>email version</th>
<th>distributed with</th>
<th>compatible with</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.x</td>
<td>Python 2.2.0 to Python 2.2.1</td>
<td>no longer supported</td>
</tr>
<tr>
<td>2.5</td>
<td>Python 2.2.2+ and Python 2.3</td>
<td>Python 2.1 to 2.5</td>
</tr>
<tr>
<td>3.0</td>
<td>Python 2.4</td>
<td>Python 2.3 to 2.5</td>
</tr>
<tr>
<td>4.0</td>
<td>Python 2.5</td>
<td>Python 2.3 to 2.5</td>
</tr>
</tbody>
</table>

Here are the major differences between email version 4 and version 3:

- All modules have been renamed according to PEP 8 standards. For example, the version 3 module `email.Message` was renamed to `email.message` in version 4.

- A new subpackage `email.mime` was added and all the version 3 `email.MIME*` modules were renamed and situated into the `email.mime` subpackage. For example, the version 3 module `email.MIMEText` was renamed to `email.mime.text`.

Note that the version 3 names will continue to work until Python 2.6.
The `email.mime.application` module was added, which contains the `MIMEApplication` class.

Methods that were deprecated in version 3 have been removed. These include `Generator.__call__()`, `Message.get_type()`, `Message.get_main_type()`, `Message.get_subtype()`.

Fixes have been added for RFC 2231 support which can change some of the return types for `Message.get_param` and friends. Under some circumstances, values which used to return a 3-tuple now return simple strings (specifically, if all extended parameter segments were unencoded, there is no language and charset designation expected, so the return type is now a simple string). Also, %-decoding used to be done for both encoded and unencoded segments; this decoding is now done only for encoded segments.

Here are the major differences between `email` version 3 and version 2:

- The `FeedParser` class was introduced, and the `Parser` class was implemented in terms of the `FeedParser`. All parsing therefore is non-strict, and parsing will make a best effort never to raise an exception. Problems found while parsing messages are stored in the message's `defect` attribute.
- All aspects of the API which raised `DeprecationWarning`s in version 2 have been removed. These include the `_encoder` argument to the `MIMEText` constructor, the `Message.add_payload()` method, the `Utils.dump_address_pair()` function, and the functions `Utils.decode()` and `Utils.encode()`.
- New `DeprecationWarning`s have been added to: `Generator.__call__()`, `Message.get_type()`, `Message.get_main_type()`, `Message.get_subtype()`, and the `strict`
argument to the `Parser` class. These are expected to be removed in future versions.

- Support for Pythons earlier than 2.3 has been removed.

Here are the differences between `email` version 2 and version 1:

- The `email.Header` and `email.Charset` modules have been added.

- The pickle format for `Message` instances has changed. Since this was never (and still isn't) formally defined, this isn't considered a backward incompatibility. However if your application pickles and unpickles `Message` instances, be aware that in `email` version 2, `Message` instances now have private variables `_charset` and `_default_type`.

- Several methods in the `Message` class have been deprecated, or their signatures changed. Also, many new methods have been added. See the documentation for the `Message` class for details. The changes should be completely backward compatible.

- The object structure has changed in the face of `message/rfc822` content types. In `email` version 1, such a type would be represented by a scalar payload, i.e. the container message's `is_multipart()` returned false, `get_payload()` was not a list object, but a single `Message` instance.

  This structure was inconsistent with the rest of the package, so the object representation for `message/rfc822` content types was changed. In `email` version 2, the container *does* return `True` from `is_multipart()`, and `get_payload()` returns a list containing a single `Message` item.

  Note that this is one place that backward compatibility could not
be completely maintained. However, if you’re already testing the return type of `get_payload()`, you should be fine. You just need to make sure your code doesn’t do a `set_payload()` with a `Message` instance on a container with a content type of `message/rfc822`.

- The `Parser` constructor’s `strict` argument was added, and its `parse()` and `parsestr()` methods grew a `headersonly` argument. The `strict` flag was also added to functions `email.message_from_file()` and `email.message_from_string()`.

- `Generator.__call__()` is deprecated; use `Generator.flatten` instead. The `Generator` class has also grown the `clone()` method.

- The `DecodedGenerator` class in the `email.generator` module was added.

- The intermediate base classes `MIMENonMultipart` and `MIMEMultipart` have been added, and interposed in the class hierarchy for most of the other MIME-related derived classes.

- The `_encoder` argument to the `MIMEText` constructor has been deprecated. Encoding now happens implicitly based on the `_charset` argument.

- The following functions in the `email.Utils` module have been deprecated: `dump_address_pairs()`, `decode()`, and `encode()`. The following functions have been added to the module: `make_msgid()`, `decode_rfc2231()`, `encode_rfc2231()`, and `decode_params()`.

- The non-public function `email.Iterator._structure()` was added.
The email package was originally prototyped as a separate library called mimelib. Changes have been made so that method names are more consistent, and some methods or modules have either been added or removed. The semantics of some of the methods have also changed. For the most part, any functionality available in mimelib is still available in the email package, albeit often in a different way. Backward compatibility between the mimelib package and the email package was not a priority.

Here is a brief description of the differences between the mimelib and the email packages, along with hints on how to port your applications.

Of course, the most visible difference between the two packages is that the package name has been changed to email. In addition, the top-level package has the following differences:

- `messageFromString()` has been renamed to `message_from_string()`.
- `messageFromFile()` has been renamed to `message_from_file()`.

The Message class has the following differences:

- The method `asString()` was renamed to `as_string()`.
- The method `ismultipart()` was renamed to `is_multipart()`.
- The `get_payload()` method has grown a `decode` optional argument.
- The method `getall()` was renamed to `get_all()`.
- The method `addheader()` was renamed to `add_header()`.
- The method `gettype()` was renamed to `get_type()`.
The method `getmaintype()` was renamed to `get_main_type()`.
The method `getsubtype()` was renamed to `get_subtype()`.
The method `getparams()` was renamed to `get_params()`. Also, whereas `getparams()` returned a list of strings, `get_params()` returns a list of 2-tuples, effectively the key/value pairs of the parameters, split on the `=' sign.
The method `getparam()` was renamed to `get_param()`.
The method `getcharsets()` was renamed to `get_charsets()`.
The method `getfilename()` was renamed to `get_filename()`.
The method `getboundary()` was renamed to `get_boundary()`.
The method `setboundary()` was renamed to `set_boundary()`.
The method `getdecodedpayload()` was removed. To get similar functionality, pass the value 1 to the `decode` flag of the `get_payload()` method.
The method `getpayloadastext()` was removed. Similar functionality is supported by the `DecodedGenerator` class in the `email.generator` module.
The method `getbodyastext()` was removed. You can get similar functionality by creating an iterator with `typed_subpart_iterator()` in the `email.iterators` module.

The `Parser` class has no differences in its public interface. It does have some additional smarts to recognize `message/delivery-status` type messages, which it represents as a `Message` instance containing separate `Message` subparts for each header block in the delivery status notification [1].

The `Generator` class has no differences in its public interface. There is a new class in the `email.generator` module though, called `DecodedGenerator` which provides most of the functionality previously available in the `Message.getpayloadastext()` method.
The following modules and classes have been changed:

- The `MIMEBase` class constructor arguments `_major` and `_minor` have changed to `_maintype` and `_subtype` respectively.

- The `Image` class/module has been renamed to `MIMEImage`. The `_minor` argument has been renamed to `_subtype`.

- The `Text` class/module has been renamed to `MIMEText`. The `_minor` argument has been renamed to `_subtype`.

- The `MessageRFC822` class/module has been renamed to `MIMEMessage`. Note that an earlier version of `mimelib` called this class/module `RFC822`, but that clashed with the Python standard library module `rfc822` on some case-insensitive file systems.

  Also, the `MIMEMessage` class now represents any kind of MIME message with main type `message`. It takes an optional argument `_subtype` which is used to set the MIME subtype. `_subtype` defaults to `rfc822`.

`mimelib` provided some utility functions in its `address` and `date` modules. All of these functions have been moved to the `email.utils` module.

The `MsgReader` class/module has been removed. Its functionality is most closely supported in the `body_line_iterator()` function in the `email.iterators` module.

**Footnotes**

[1] Delivery Status Notifications (DSN) are defined in RFC 1894.
18.1.1. `email.message`: Representing an email message

The central class in the `email` package is the `Message` class, imported from the `email.message` module. It is the base class for the `email` object model. `Message` provides the core functionality for setting and querying header fields, and for accessing message bodies.

Conceptually, a `Message` object consists of `headers` and `payloads`. Headers are RFC 2822 style field names and values where the field name and value are separated by a colon. The colon is not part of either the field name or the field value.

Headers are stored and returned in case-preserving form but are matched case-insensitively. There may also be a single envelope header, also known as the Unix-From header or the From_ header. The payload is either a string in the case of simple message objects or a list of `Message` objects for MIME container documents (e.g. `multipart/*` and `message/rfc822`).

`Message` objects provide a mapping style interface for accessing the message headers, and an explicit interface for accessing both the headers and the payload. It provides convenience methods for generating a flat text representation of the message object tree, for accessing commonly used header parameters, and for recursively walking over the object tree.

Here are the methods of the `Message` class:

```python
class email.message.Message
    The constructor takes no arguments.
    as_string([unixfrom])
```
Return the entire message flattened as a string. When optional `unixfrom` is `True`, the envelope header is included in the returned string. `unixfrom` defaults to `False`. Flattening the message may trigger changes to the `Message` if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it mangles lines that begin with `From`. For more flexibility, instantiate a `Generator` instance and use its `flatten()` method directly. For example:

```python
from cStringIO import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=False, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

`__str__()`
Equivalent to `as_string(unixfrom=True)`.

`is_multipart()`
Return `True` if the message's payload is a list of sub-`Message` objects, otherwise return `False`. When `is_multipart()` returns `False`, the payload should be a string object.

`set_unixfrom(unixfrom)`
Set the message’s envelope header to `unixfrom`, which should be a string.

`get_unixfrom()`
Return the message’s envelope header. Defaults to `None` if the envelope header was never set.
attach(payload)
Add the given payload to the current payload, which must be None or a list of Message objects before the call. After the call, the payload will always be a list of Message objects. If you want to set the payload to a scalar object (e.g. a string), use set_payload() instead.

get_payload([i, decode])
Return the current payload, which will be a list of Message objects when is_multipart() is True, or a string when is_multipart() is False. If the payload is a list and you mutate the list object, you modify the message’s payload in place.

With optional argument i, get_payload() will return the i-th element of the payload, counting from zero, if is_multipart() is True. An IndexError will be raised if i is less than 0 or greater than or equal to the number of items in the payload. If the payload is a string (i.e. is_multipart() is False) and i is given, a TypeError is raised.

Optional decode is a flag indicating whether the payload should be decoded or not, according to the Content-Transfer-Encoding header. When True and the message is not a multipart, the payload will be decoded if this header’s value is quoted-printable or base64. If some other encoding is used, or Content-Transfer-Encoding header is missing, or if the payload has bogus base64 data, the payload is returned as-is (undecoded). If the message is a multipart and the decode flag is True, then None is returned. The default for decode is False.

set_payload(payload[, charset])
Set the entire message object’s payload to \textit{payload}. It is the client’s responsibility to ensure the payload invariants. Optional \texttt{charset} sets the message’s default character set; see \texttt{set_charset()} for details.

\textit{Changed in version 2.2.2: charset argument added.}

\texttt{set charset(charset)}

Set the character set of the payload to \texttt{charset}, which can either be a \texttt{Charset} instance (see \texttt{email.charset}), a string naming a character set, or \texttt{None}. If it is a string, it will be converted to a \texttt{Charset} instance. If \texttt{charset} is \texttt{None}, the \texttt{charset} parameter will be removed from the \texttt{Content-Type} header (the message will not be otherwise modified). Anything else will generate a \texttt{TypeError}.

If there is no existing \texttt{MIME-Version} header one will be added. If there is no existing \texttt{Content-Type} header, one will be added with a value of \texttt{text/plain}. Whether the \texttt{Content-Type} header already exists or not, its \texttt{charset} parameter will be set to \texttt{charset.output_charset}. If \texttt{charset.input_charset} and \texttt{charset.output_charset} differ, the payload will be re-encoded to the \texttt{output_charset}. If there is no existing \texttt{Content-Transfer-Encoding} header, then the payload will be transfer-encoded, if needed, using the specified \texttt{Charset}, and a header with the appropriate value will be added. If a \texttt{Content-Transfer-Encoding} header already exists, the payload is assumed to already be correctly encoded using that \texttt{Content-Transfer-Encoding} and is not modified.

The message will be assumed to be of type \texttt{text/*}, with the payload either in unicode or encoded with \texttt{charset.input_charset}. It will be encoded or converted to \texttt{charset.output_charset} and transfer encoded properly, if
needed, when generating the plain text representation of the message. MIME headers \((MIME-Version, Content-Type, Content-Transfer-Encoding)\) will be added as needed.

*New in version 2.2.2.*

**get_charset()**

Return the `charset` instance associated with the message’s payload.

*New in version 2.2.2.*

The following methods implement a mapping-like interface for accessing the message’s RFC 2822 headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by `keys()`, but in a `Message` object, headers are always returned in the order they appeared in the original message, or were added to the message later. Any header deleted and then re-added are always appended to the end of the header list.

These semantic differences are intentional and are biased toward maximal convenience.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

**__len__()**

Return the total number of headers, including duplicates.

**__contains__(name)**

Return true if the message object has a field named `name`. 
Matching is done case-insensitively and name should not include the trailing colon. Used for the in operator, e.g.:

```python
if 'message-id' in myMessage:
    print 'Message-ID:', myMessage['message-id']
```

__getitem__(name)

Return the value of the named header field. name should not include the colon field separator. If the header is missing, None is returned; a KeyError is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the get_all() method to get the values of all the extant named headers.

__setitem__(name, val)

Add a header to the message with field name name and value val. The field is appended to the end of the message’s existing fields.

Note that this does not overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name name, delete the field first, e.g.:

```python
del msg['subject']
msg['subject'] = 'Python roolz!'```

__delitem__(name)

Delete all occurrences of the field with name name from the message's headers. No exception is raised if the named field isn't present in the headers.

has_key(name)

Return true if the message contains a header field named
name, otherwise return false.

keys()
Return a list of all the message's header field names.

values()
Return a list of all the message's field values.

items()
Return a list of 2-tuples containing all the message's field headers and values.

get(name[, failobj])
Return the value of the named header field. This is identical to __getitem__() except that optional failobj is returned if the named header is missing (defaults to None).

Here are some additional useful methods:

get_all(name[, failobj])
Return a list of all the values for the field named name. If there are no such named headers in the message, failobj is returned (defaults to None).

add_header(_name, _value, **_params)
Extended header setting. This method is similar to __setitem__() except that additional header parameters can be provided as keyword arguments. _name is the header field to add and _value is the primary value for the header.

For each item in the keyword argument dictionary _params, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as
key="value" unless the value is None, in which case only the key will be added. If the value contains non-ASCII characters, it must be specified as a three tuple in the format (CHARSET, LANGUAGE, VALUE), where CHARSET is a string naming the charset to be used to encode the value, LANGUAGE can usually be set to None or the empty string (see RFC 2231 for other possibilities), and VALUE is the string value containing non-ASCII code points.

Here's an example:

```python
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

An example with non-ASCII characters:

```python
msg.add_header('Content-Disposition', 'attachment', filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

Which produces

```
Content-Disposition: attachment; filename*="iso-8859-1''Fu%DFballer.ppt"
```

**replace_header(_name, _value)**

Replace a header. Replace the first header found in the message that matches _name, retaining header order and field name case. If no matching header was found, a KeyError is raised.

*New in version 2.2.2.*
get_content_type()
Return the message's content type. The returned string is coerced to lower case of the form maintype/subtype. If there was no Content-Type header in the message the default type as given by get_default_type() will be returned. Since according to RFC 2045, messages always have a default type, get_content_type() will always return a value.

RFC 2045 defines a message's default type to be text/plain unless it appears inside a multipart/digest container, in which case it would be message/rfc822. If the Content-Type header has an invalid type specification, RFC 2045 mandates that the default type be text/plain.

New in version 2.2.2.

get_content_maintype()
Return the message's main content type. This is the maintype part of the string returned by get_content_type().

New in version 2.2.2.

get_content_subtype()
Return the message's sub-content type. This is the subtype part of the string returned by get_content_type().

New in version 2.2.2.

get_default_type()
Return the default content type. Most messages have a default content type of text/plain, except for messages that are subparts of multipart/digest containers. Such subparts have a default content type of message/rfc822.

New in version 2.2.2.
set_default_type(
 ctype)
 Set the default content type. ctype should either be text/plain or message/rfc822, although this is not enforced. The default content type is not stored in the Content-Type header.

New in version 2.2.2.

get_params([failobj[, header[, unquote]]])
 Return the message's Content-Type parameters, as a list. The elements of the returned list are 2-tuples of key/value pairs, as split on the '=' sign. The left hand side of the '=' is the key, while the right hand side is the value. If there is no '=' sign in the parameter the value is the empty string, otherwise the value is as described in get_param() and is unquoted if optional unquote is True (the default).

Optional failobj is the object to return if there is no Content-Type header. Optional header is the header to search instead of Content-Type.

Changed in version 2.2.2: unquote argument added.

get_param(param[, failobj[, header[, unquote]]])
 Return the value of the Content-Type header's parameter param as a string. If the message has no Content-Type header or if there is no such parameter, then failobj is returned (defaults to None).

Optional header if given, specifies the message header to use instead of Content-Type.

Parameter keys are always compared case insensitively. The return value can either be a string, or a 3-tuple if the parameter was RFC 2231 encoded. When it's a 3-tuple, the
elements of the value are of the form \((\text{CHARSET, LANGUAGE, VALUE})\). Note that both \texttt{CHARSET} and \texttt{LANGUAGE} can be \texttt{None}, in which case you should consider \texttt{VALUE} to be encoded in the \texttt{us-ascii} charset. You can usually ignore \texttt{LANGUAGE}.

If your application doesn’t care whether the parameter was encoded as in \textbf{RFC 2231}, you can collapse the parameter value by calling \texttt{email.utils.collapse_rfc2231_value()}, passing in the return value from \texttt{get_param()}. This will return a suitably decoded Unicode string when the value is a tuple, or the original string unquoted if it isn’t. For example:

```python
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

In any case, the parameter value (either the returned string, or the \texttt{VALUE} item in the 3-tuple) is always unquoted, unless \texttt{unquote} is set to \texttt{False}.

\textit{Changed in version 2.2.2: unquote argument added, and 3-tuple return value possible.}

\texttt{set_param(param, value[, header[, requote[, charset[, language]]]])}

Set a parameter in the \texttt{Content-Type} header. If the parameter already exists in the header, its value will be replaced with \texttt{value}. If the \texttt{Content-Type} header as not yet been defined for this message, it will be set to \texttt{text/plain} and the new parameter value will be appended as per \textbf{RFC 2045}.

Optional \texttt{header} specifies an alternative header to \texttt{Content-Type}, and all parameters will be quoted as necessary unless optional \texttt{requote} is \texttt{False} (the default is \texttt{True}).
If optional `charset` is specified, the parameter will be encoded according to RFC 2231. Optional `language` specifies the RFC 2231 language, defaulting to the empty string. Both `charset` and `language` should be strings.

*New in version 2.2.2.*

`del_param(param[, header[, requote]])`
Remove the given parameter completely from the Content-Type header. The header will be re-written in place without the parameter or its value. All values will be quoted as necessary unless `requote` is `False` (the default is `True`). Optional `header` specifies an alternative to Content-Type.

*New in version 2.2.2.*

`set_type(type[, header][, requote])`
Set the main type and subtype for the Content-Type header. `type` must be a string in the form `maintype/subtype`, otherwise a `ValueError` is raised.

This method replaces the Content-Type header, keeping all the parameters in place. If `requote` is `False`, this leaves the existing header’s quoting as is, otherwise the parameters will be quoted (the default).

An alternative header can be specified in the `header` argument. When the Content-Type header is set a MIME-Version header is also added.

*New in version 2.2.2.*

`get_filename([failobj])`
Return the value of the `filename` parameter of the Content-Disposition header of the message. If the header does not
have a filename parameter, this method falls back to looking for the name parameter on the Content-Type header. If neither is found, or the header is missing, then failobj is returned. The returned string will always be unquoted as per email.utils.unquote().

**get_boundary**([failobj])

Return the value of the boundary parameter of the Content-Type header of the message, or failobj if either the header is missing, or has no boundary parameter. The returned string will always be unquoted as per email.utils.unquote().

**set_boundary**(boundary)

Set the boundary parameter of the Content-Type header to boundary. set_boundary() will always quote boundary if necessary. A HeaderParseError is raised if the message object has no Content-Type header.

Note that using this method is subtly different than deleting the old Content-Type header and adding a new one with the new boundary via add_header(), because set_boundary() preserves the order of the Content-Type header in the list of headers. However, it does not preserve any continuation lines which may have been present in the original Content-Type header.

**get_content_charset**([failobj])

Return the charset parameter of the Content-Type header, coerced to lower case. If there is no Content-Type header, or if that header has no charset parameter, failobj is returned.

Note that this method differs from get_charset() which returns the Charset instance for the default encoding of the
message body.

*New in version 2.2.2.*

def get_charsets([failobj])
    Return a list containing the character set names in the message. If the message is a *multipart*, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

    Each item in the list will be a string which is the value of the *charset* parameter in the *Content-Type* header for the represented subpart. However, if the subpart has no *Content-Type* header, no *charset* parameter, or is not of the *text* main MIME type, then that item in the returned list will be *failobj*.

*walk()*

    The *walk()* method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use *walk()* as the iterator in a *for* loop; each iteration returns the next subpart.

    Here’s an example that prints the MIME type of every part of a multipart message structure:

    ```
    >>> for part in msg.walk():
    ...     print part.get_content_type()
    multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
    ```

*Changed in version 2.5:* The previously deprecated methods *get_type()* , *get_main_type()* , and *get_subtype()* were removed.
Message objects can also optionally contain two instance attributes, which can be used when generating the plain text of a MIME message.

preamble

The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The preamble attribute contains this leading extra-armor text for MIME documents. When the Parser discovers some text after the headers but before the first boundary string, it assigns this text to the message's preamble attribute. When the Generator is writing out the plain text representation of a MIME message, and it finds the message has a preamble attribute, it will write this text in the area between the headers and the first boundary. See email.parser and email.generator for details.

Note that if the message object has no preamble, the preamble attribute will be None.

epilogue

The epilogue attribute acts the same way as the preamble attribute, except that it contains text that appears between the last boundary and the end of the message.

Changed in version 2.5: You do not need to set the epilogue to the empty string in order for the Generator to print a newline at the end of the file.
defects

The *defects* attribute contains a list of all the problems found when parsing this message. See `email.errors` for a detailed description of the possible parsing defects.

*New in version 2.4.*
18.1. `email` — An email and MIME handling package
18.1.2. email.parser: Parsing email messages

Message object structures can be created in one of two ways: they can be created from whole cloth by instantiating `Message` objects and stringing them together via `attach()` and `set_payload()` calls, or they can be created by parsing a flat text representation of the email message.

The `email` package provides a standard parser that understands most email document structures, including MIME documents. You can pass the parser a string or a file object, and the parser will return to you the root `Message` instance of the object structure. For simple, non-MIME messages the payload of this root object will likely be a string containing the text of the message. For MIME messages, the root object will return `True` from its `is_multipart()` method, and the subparts can be accessed via the `get_payload()` and `walk()` methods.

There are actually two parser interfaces available for use, the classic `Parser` API and the incremental `FeedParser` API. The classic `Parser` API is fine if you have the entire text of the message in memory as a string, or if the entire message lives in a file on the file system. `FeedParser` is more appropriate for when you’re reading the message from a stream which might block waiting for more input (e.g. reading an email message from a socket). The `FeedParser` can consume and parse the message incrementally, and only returns the root object when you close the parser [1].

Note that the parser can be extended in limited ways, and of course you can implement your own parser completely from scratch. There is no magical connection between the `email` package's bundled
parser and the `Message` class, so your custom parser can create message object trees any way it finds necessary.
18.1.2.1. FeedParser API

New in version 2.4.

The FeedParser, imported from the email.feedparser module, provides an API that is conducive to incremental parsing of email messages, such as would be necessary when reading the text of an email message from a source that can block (e.g. a socket). The FeedParser can of course be used to parse an email message fully contained in a string or a file, but the classic Parser API may be more convenient for such use cases. The semantics and results of the two parser APIs are identical.

The FeedParser's API is simple; you create an instance, feed it a bunch of text until there's no more to feed it, then close the parser to retrieve the root message object. The FeedParser is extremely accurate when parsing standards-compliant messages, and it does a very good job of parsing non-compliant messages, providing information about how a message was deemed broken. It will populate a message object's defects attribute with a list of any problems it found in a message. See the email.errors module for the list of defects that it can find.

Here is the API for the FeedParser:

```python
class email.parser.FeedParser(_factory)

    Create a FeedParser instance. Optional _factory is a no-argument
    callable that will be called whenever a new message object is
    needed. It defaults to the email.message.Message class.

    feed(data)

    Feed the FeedParser some more data. data should be a string
```
containing one or more lines. The lines can be partial and the FeedParser will stitch such partial lines together properly. The lines in the string can have any of the common three line endings, carriage return, newline, or carriage return and newline (they can even be mixed).

close()

Closing a FeedParser completes the parsing of all previously fed data, and returns the root message object. It is undefined what happens if you feed more data to a closed FeedParser.
18.1.2.2. Parser class API

The **Parser** class, imported from the `email.parser` module, provides
an API that can be used to parse a message when the complete
contents of the message are available in a string or file. The
`email.parser` module also provides a second class, called
**HeaderParser** which can be used if you're only interested in the
headers of the message. **HeaderParser** can be much faster in these
situations, since it does not attempt to parse the message body,
instead setting the payload to the raw body as a string. **HeaderParser**
has the same API as the **Parser** class.

```python
class email.parser.Parser(_class)
```

The constructor for the **Parser** class takes an optional argument
`_class`. This must be a callable factory (such as a function or a
class), and it is used whenever a sub-message object needs to
be created. It defaults to **Message** (see `email.message`). The
factory will be called without arguments.

The optional **strict** flag is ignored.

---

*Deprecated since version 2.4:* Because the **Parser** class is a
backward compatible API wrapper around the new-in-Python
2.4 **FeedParser**, all parsing is effectively non-strict. You should
simply stop passing a **strict** flag to the **Parser** constructor.

*Changed in version 2.2.2:* The **strict** flag was added.

*Changed in version 2.4:* The **strict** flag was deprecated.

The other public **Parser** methods are:

```python
parse(fp[, headersonly])
```
Read all the data from the file-like object \textit{fp}, parse the resulting text, and return the root message object. \textit{fp} must support both the \texttt{readline()} and the \texttt{read()} methods on file-like objects.

The text contained in \textit{fp} must be formatted as a block of RFC 2822 style headers and header continuation lines, optionally preceded by an envelope header. The header block is terminated either by the end of the data or by a blank line. Following the header block is the body of the message (which may contain MIME-encoded subparts).

Optional \texttt{headersonly} is a flag specifying whether to stop parsing after reading the headers or not. The default is \texttt{False}, meaning it parses the entire contents of the file.

\textit{Changed in version 2.2.2:} The \texttt{headersonly} flag was added.

\textbf{parsestr}(\texttt{text}[\texttt{, headersonly}])

Similar to the \texttt{parse()} method, except it takes a string object instead of a file-like object. Calling this method on a string is exactly equivalent to wrapping \texttt{text} in a \texttt{StringIO} instance first and calling \texttt{parse()}.

Optional \texttt{headersonly} is as with the \texttt{parse()} method.

\textit{Changed in version 2.2.2:} The \texttt{headersonly} flag was added.

Since creating a message object structure from a string or a file object is such a common task, two functions are provided as a convenience. They are available in the top-level \texttt{email} package namespace.

\texttt{email.message_from_string(s[, \_class[, strict]]})
Return a message object structure from a string. This is exactly equivalent to `Parser().parsestr(s)`. Optional `_class` and `strict` are interpreted as with the `Parser` class constructor.

*Changed in version 2.2.2:* The `strict` flag was added.

```
e-mail.message_from_file(fp[, _class[, strict]])
```

Return a message object structure tree from an open file object. This is exactly equivalent to `Parser().parse(fp)`. Optional `_class` and `strict` are interpreted as with the `Parser` class constructor.

*Changed in version 2.2.2:* The `strict` flag was added.

Here’s an example of how you might use this at an interactive Python prompt:

```
>>> import email
>>> msg = email.message_from_string(myString)
```
18.1.2.3. Additional notes

Here are some notes on the parsing semantics:

- Most non-`multipart` type messages are parsed as a single message object with a string payload. These objects will return `False` for `is_multipart()`. Their `get_payload()` method will return a string object.
- All `multipart` type messages will be parsed as a container message object with a list of sub-message objects for their payload. The outer container message will return `True` for `is_multipart()` and their `get_payload()` method will return the list of `Message` subparts.
- Most messages with a content type of `message/*` (e.g. `message/delivery-status` and `message/rfc822`) will also be parsed as container object containing a list payload of length 1. Their `is_multipart()` method will return `True`. The single element in the list payload will be a sub-message object.
- Some non-standards compliant messages may not be internally consistent about their `multipart`-edness. Such messages may have a `Content-Type` header of type `multipart`, but their `is_multipart()` method may return `False`. If such messages were parsed with the `FeedParser`, they will have an instance of the `MultipartInvariantViolationDefect` class in their `defects` attribute list. See `email.errors` for details.

Footnotes

[1] As of email package version 3.0, introduced in Python 2.4, the classic `Parser` was re-implemented in terms of the `FeedParser`, so the semantics and results are identical between the two parsers.
18.1.3. `email.generator`: Generating MIME documents

One of the most common tasks is to generate the flat text of the email message represented by a message object structure. You will need to do this if you want to send your message via the `smtplib` module or the `nntplib` module, or print the message on the console. Taking a message object structure and producing a flat text document is the job of the `Generator` class.

Again, as with the `email.parser` module, you aren’t limited to the functionality of the bundled generator; you could write one from scratch yourself. However the bundled generator knows how to generate most email in a standards-compliant way, should handle MIME and non-MIME email messages just fine, and is designed so that the transformation from flat text, to a message structure via the `Parser` class, and back to flat text, is idempotent (the input is identical to the output) [1]. On the other hand, using the Generator on a `Message` constructed by program may result in changes to the `Message` object as defaults are filled in.

Here are the public methods of the `Generator` class, imported from the `email.generator` module:

```python
class email.generator.Generator(outfp[, mangle_from_[, maxheaderlen]])
```

The constructor for the `Generator` class takes a file-like object called `outfp` for an argument. `outfp` must support the `write()` method and be usable as the output file in a Python extended print statement.
Optional `mangle_from_` is a flag that, when `True`, puts a `>` character in front of any line in the body that starts exactly as `From`, i.e. `From` followed by a space at the beginning of the line. This is the only guaranteed portable way to avoid having such lines be mistaken for a Unix mailbox format envelope header separator (see [WHY THE CONTENT-LENGTH FORMAT IS BAD](#) for details). `mangle_from_` defaults to `True`, but you might want to set this to `False` if you are not writing Unix mailbox format files.

Optional `maxheaderlen` specifies the longest length for a non-continued header. When a header line is longer than `maxheaderlen` (in characters, with tabs expanded to 8 spaces), the header will be split as defined in the `Header` class. Set to zero to disable header wrapping. The default is 78, as recommended (but not required) by [RFC 2822](https://tools.ietf.org/html/rfc2822).

The other public `Generator` methods are:

### flatten(msg[, unixfrom])

Print the textual representation of the message object structure rooted at `msg` to the output file specified when the `Generator` instance was created. Subparts are visited depth-first and the resulting text will be properly MIME encoded.

Optional `unixfrom` is a flag that forces the printing of the envelope header delimiter before the first [RFC 2822](https://tools.ietf.org/html/rfc2822) header of the root message object. If the root object has no envelope header, a standard one is crafted. By default, this is set to `False` to inhibit the printing of the envelope delimiter.

Note that for subparts, no envelope header is ever printed.

New in version 2.2.2.
**clone**(*fp*)

Return an independent clone of this **Generator** instance with the exact same options.

*New in version 2.2.2.*

**write**(*s*)

Write the string *s* to the underlying file object, i.e. *outfp* passed to **Generator**'s constructor. This provides just enough file-like API for **Generator** instances to be used in extended print statements.

As a convenience, see the methods **Message.as_string()** and **str(aMessage)**, a.k.a. **Message.__str__()**, which simplify the generation of a formatted string representation of a message object. For more detail, see **email.message**.

The **email.generator** module also provides a derived class, called **DecodedGenerator** which is like the **Generator** base class, except that non-**text** parts are substituted with a format string representing the part.

```python
class email.generator.**DecodedGenerator**(*outfp*, mangle_from_, maxheaderlen[, fmt])
```

This class, derived from **Generator** walks through all the subparts of a message. If the subpart is of main type **text**, then it prints the decoded payload of the subpart. Optional `_mangle_from_` and `maxheaderlen` are as with the **Generator** base class.

If the subpart is not of main type **text**, optional *fmt* is a format string that is used instead of the message payload. *fmt* is expanded with the following keywords, `%(keyword)s` format:
The default value for fmt is None, meaning

Non-text (%(type)s) part of message omitted, filename %(filename)s

New in version 2.2.2.

Changed in version 2.5: The previously deprecated method __call__() was removed.

Footnotes

[1] This statement assumes that you use the appropriate setting for the unixfrom argument, and that you set maxheaderlen=0 (which will preserve whatever the input line lengths were). It is also not strictly true, since in many cases runs of whitespace in headers are collapsed into single blanks. The latter is a bug that will eventually be fixed.
18.1.4. email.mime: Creating email and MIME objects from scratch

Ordinarily, you get a message object structure by passing a file or some text to a parser, which parses the text and returns the root message object. However you can also build a complete message structure from scratch, or even individual Message objects by hand. In fact, you can also take an existing structure and add new Message objects, move them around, etc. This makes a very convenient interface for slicing-and-dicing MIME messages.

You can create a new object structure by creating Message instances, adding attachments and all the appropriate headers manually. For MIME messages though, the email package provides some convenient subclasses to make things easier.

Here are the classes:

class email.mime.base.MIMEBase(_maintype, _subtype, **_params)

Module: email.mime.base

This is the base class for all the MIME-specific subclasses of Message. Ordinarily you won’t create instances specifically of MIMEBase, although you could. MIMEBase is provided primarily as a convenient base class for more specific MIME-aware subclasses.

_maintype is the Content-Type major type (e.g. text or image), and _subtype is the Content-Type minor type (e.g. plain or gif). _params is a parameter key/value dictionary and is passed directly to Message.add_header.

The MIMEBase class always adds a Content-Type header (based on _maintype, _subtype, and _params), and a MIME-Version
A subclass of MIMEBase, this is an intermediate base class for MIME messages that are not multipart. The primary purpose of this class is to prevent the use of the attach() method, which only makes sense for multipart messages. If attach() is called, a MultipartConversionError exception is raised.

New in version 2.2.2.

A subclass of MIMEBase, this is an intermediate base class for MIME messages that are multipart. Optional _subtype defaults to mixed, but can be used to specify the subtype of the message. A Content-Type header of multipart/_subtype will be added to the message object. A MIME-Version header will also be added.

Optional boundary is the multipart boundary string. When None (the default), the boundary is calculated when needed (for example, when the message is serialized).

_subparts is a sequence of initial subparts for the payload. It must be possible to convert this sequence to a list. You can always attach new subparts to the message by using the Message.attach method.

Additional parameters for the Content-Type header are taken from the keyword arguments, or passed into the _params
argument, which is a keyword dictionary.

_New in version 2.2.2._

class email.mime.application.\texttt{MIMEApplication}(_\texttt{data}[,_\texttt{subtype}[,_\texttt{encoder}[,**_\texttt{params}]]])

Module: \texttt{email.mime.application}

A subclass of \texttt{MIMENonMultipart}, the \texttt{MIMEApplication} class is used to represent MIME message objects of major type \textit{application}. \_data is a string containing the raw byte data. Optional \_subtype specifies the MIME subtype and defaults to \textit{octet-stream}.

Optional \_encoder is a callable (i.e. function) which will perform the actual encoding of the data for transport. This callable takes one argument, which is the \texttt{MIMEApplication} instance. It should use \texttt{get\_payload()} and \texttt{set\_payload()} to change the payload to encoded form. It should also add any \textit{Content-Transfer-Encoding} or other headers to the message object as necessary. The default encoding is base64. See the \texttt{email.encoders} module for a list of the built-in encoders.

\_params are passed straight through to the base class constructor.

_New in version 2.5._

class email.mime.audio.\texttt{MIMEAudio}(_\texttt{audiodata}[,_\texttt{subtype}[,_\texttt{encoder}[,**_\texttt{params}]]])

Module: \texttt{email.mime.audio}

A subclass of \texttt{MIMENonMultipart}, the \texttt{MIMEAudio} class is used to create MIME message objects of major type \textit{audio}. \_audiodata is
a string containing the raw audio data. If this data can be decoded by the standard Python module \texttt{sndhdr}, then the subtype will be automatically included in the \textit{Content-Type} header. Otherwise you can explicitly specify the audio subtype via the \_\texttt{subtype} parameter. If the minor type could not be guessed and \_\texttt{subtype} was not given, then \texttt{TypeError} is raised.

Optional \_\texttt{encoder} is a callable (i.e. function) which will perform the actual encoding of the audio data for transport. This callable takes one argument, which is the \texttt{MIMEAudio} instance. It should use \texttt{get\_payload()} and \texttt{set\_payload()} to change the payload to encoded form. It should also add any \textit{Content-Transfer-Encoding} or other headers to the message object as necessary. The default encoding is base64. See the \texttt{email.encoders} module for a list of the built-in encoders.

\_\texttt{params} are passed straight through to the base class constructor.

\texttt{class email.mime.image.MIMEImage(_imagedata[], _subtype[], _encoder[], **_params[]])}

Module: \texttt{email.mime.image}

A subclass of \texttt{MIMENonMultipart}, the \texttt{MIMEImage} class is used to create MIME message objects of major type \textit{image}. \_\texttt{imagedata} is a string containing the raw image data. If this data can be decoded by the standard Python module \texttt{imghdr}, then the subtype will be automatically included in the \textit{Content-Type} header. Otherwise you can explicitly specify the image subtype via the \_\texttt{subtype} parameter. If the minor type could not be guessed and \_\texttt{subtype} was not given, then \texttt{TypeError} is raised.

Optional \_\texttt{encoder} is a callable (i.e. function) which will perform the actual encoding of the image data for transport. This callable
takes one argument, which is the `MIMEImage` instance. It should use `get_payload()` and `set_payload()` to change the payload to encoded form. It should also add any `Content-Transfer-Encoding` or other headers to the message object as necessary. The default encoding is base64. See the `email.encoders` module for a list of the built-in encoders.

`_params` are passed straight through to the `MIMEBase` constructor.

class email.mime.message.MIMEMessage(_msg[, _subtype])
Module: email.mime.message

A subclass of `MIMENonMultipart`, the `MIMEMessage` class is used to create MIME objects of main type `message`. `_msg` is used as the payload, and must be an instance of class `Message` (or a subclass thereof), otherwise a `TypeError` is raised.

Optional `_subtype` sets the subtype of the message; it defaults to `rfc822`.

class email.mime.text.MIMEText(_text[, _subtype[, _charset]])
Module: email.mime.text

A subclass of `MIMENonMultipart`, the `MIMEText` class is used to create MIME objects of major type `text`. `_text` is the string for the payload. `_subtype` is the minor type and defaults to `plain`. `_charset` is the character set of the text and is passed as a parameter to the `MIMENonMultipart` constructor; it defaults to `us-ascii`. If `_text` is unicode, it is encoded using the `output_charset` of `_charset`, otherwise it is used as-is.

*Changed in version 2.4:* The previously deprecated `_encoding` argument has been removed. Content Transfer Encoding now happens implicitly based on the `_charset` argument.
Unless the `_charset` parameter is explicitly set to `None`, the 
MIMEText object created will have both a `Content-Type` header 
with a `charset` parameter, and a `Content-Transfer-Encoding` 
header. This means that a subsequent `set_payload` call will not 
result in an encoded payload, even if a charset is passed in the 
`set_payload` command. You can “reset” this behavior by deleting 
the `Content-Transfer-Encoding` header, after which a `set_payload` 
call will automatically encode the new payload (and add a new 
`Content-Transfer-Encoding` header).
18. Internet Data Handling

18.1. email — An email and MIME handling package
18.1.5. email.header: Internationalized headers

RFC 2822 is the base standard that describes the format of email messages. It derives from the older RFC 822 standard which came into widespread use at a time when most email was composed of ASCII characters only. RFC 2822 is a specification written assuming email contains only 7-bit ASCII characters.

Of course, as email has been deployed worldwide, it has become internationalized, such that language specific character sets can now be used in email messages. The base standard still requires email messages to be transferred using only 7-bit ASCII characters, so a slew of RFCs have been written describing how to encode email containing non-ASCII characters into RFC 2822-compliant format. These RFCs include RFC 2045, RFC 2046, RFC 2047, and RFC 2231. The email package supports these standards in its email.header and email.charset modules.

If you want to include non-ASCII characters in your email headers, say in the Subject or To fields, you should use the Header class and assign the field in the Message object to an instance of Header instead of using a string for the header value. Import the Header class from the email.header module. For example:

```python
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('\uf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> print(msg.as_string())
Subject: =?iso-8859-1?q?p=F6stal?
```

Notice here how we wanted the Subject field to contain a non-ASCII
character? We did this by creating a `Header` instance and passing in the character set that the byte string was encoded in. When the subsequent `Message` instance was flattened, the `Subject` field was properly RFC 2047 encoded. MIME-aware mail readers would show this header using the embedded ISO-8859-1 character.

New in version 2.2.2.

Here is the `Header` class description:

```python
class email.header.Header([s[, charset[, maxlinelen[, 
header_name[, continuation_ws[, errors]]]]]])
```

Create a MIME-compliant header that can contain strings in different character sets.

Optional `s` is the initial header value. If `None` (the default), the initial header value is not set. You can later append to the header with `append()` method calls. `s` may be a byte string or a Unicode string, but see the `append()` documentation for semantics.

Optional `charset` serves two purposes: it has the same meaning as the `charset` argument to the `append()` method. It also sets the default character set for all subsequent `append()` calls that omit the `charset` argument. If `charset` is not provided in the constructor (the default), the `us-ascii` character set is used both as `s`'s initial charset and as the default for subsequent `append()` calls.

The maximum line length can be specified explicitly via `maxlinelen`. For splitting the first line to a shorter value (to account for the field header which isn’t included in `s`, e.g. `Subject`) pass in the name of the field in `header_name`. The default `maxlinelen` is 76, and the default value for `header_name` is `None`, meaning it is not taken into account for the first line of a long, split header.
Optional `continuation_ws` must be RFC 2822-compliant folding whitespace, and is usually either a space or a hard tab character. This character will be prepended to continuation lines. `continuation_ws` defaults to a single space character (" ").

Optional `errors` is passed straight through to the `append()` method.

```
append(s[, charset[, errors]])
```

Append the string `s` to the MIME header.

Optional `charset`, if given, should be a `Charset` instance (see `email.charset`) or the name of a character set, which will be converted to a `Charset` instance. A value of `None` (the default) means that the `charset` given in the constructor is used.

`s` may be a byte string or a Unicode string. If it is a byte string (i.e. `isinstance(s, str)` is true), then `charset` is the encoding of that byte string, and a `UnicodeError` will be raised if the string cannot be decoded with that character set.

If `s` is a Unicode string, then `charset` is a hint specifying the character set of the characters in the string. In this case, when producing an RFC 2822-compliant header using RFC 2047 rules, the Unicode string will be encoded using the following charsets in order: `us-ascii`, the `charset` hint, `utf-8`. The first character set to not provoke a `UnicodeError` is used.

Optional `errors` is passed through to any `unicode()` or `unicode.encode()` call, and defaults to “strict”.

```
encode([splitchars])
```

Encode a message header into an RFC-compliant format, possibly wrapping long lines and encapsulating non-ASCII
parts in base64 or quoted-printable encodings. Optional
`splitchars` is a string containing characters to split long ASCII
lines on, in rough support of RFC 2822’s highest level
syntactic breaks. This doesn’t affect RFC 2047 encoded lines.

The `Header` class also provides a number of methods to support
standard operators and built-in functions.

```python
__str__()
    A synonym for `Header.encode()`. Useful for `str(aHeader)`.

__unicode__()
    A helper for the built-in `unicode()` function. Returns the
    header as a Unicode string.

__eq__(other)
    This method allows you to compare two `Header` instances for
    equality.

__ne__(other)
    This method allows you to compare two `Header` instances for
    inequality.
```

The `email.header` module also provides the following convenient
functions.

```python
email.header. decode_header(header)
    Decode a message header value without converting the
    character set. The header value is in `header`.
```

This function returns a list of `(decoded_string, charset)` pairs
containing each of the decoded parts of the header. `charset` is
`None` for non-encoded parts of the header, otherwise a lower case
string containing the name of the character set specified in the
encoded string.
Here’s an example:
>>> from email.header import decode_header
[('p\xf6stal', 'iso-8859-1')]

[, maxlinelen[,

email.header. make_header (decoded_seq

header_name[, continuation_ws]]])
Create a Header instance from a sequence of pairs as returned
by decode_header() .
takes a header value string and returns a
sequence of pairs of the format (decoded_string, charset)
where charset is the name of the character set.
decode_header()

This function takes one of those sequence of pairs and returns a
Header instance. Optional maxlinelen, header_name, and
continuation_ws are as in the Header constructor.

Python » Python 2.7.13 documentation » The Python

email

previous | next | modules | index
— An email and MIME handling

package »
© Copyright 1990-2016, Python Software Foundation.
The Python Software Foundation is a non-profit corporation. Please donate.
Last updated on Dec 17, 2016. Found a bug?
Created using Sphinx 1.4.4.


18.1.6. `email.charset`: Representing character sets

This module provides a class `Charset` for representing character sets and character set conversions in email messages, as well as a character set registry and several convenience methods for manipulating this registry. Instances of `Charset` are used in several other modules within the `email` package.

Import this class from the `email.charset` module.

*New in version 2.2.2.*

```python
class email.charset.Charset([input_charset])
```

Map character sets to their email properties.

This class provides information about the requirements imposed on email for a specific character set. It also provides convenience routines for converting between character sets, given the availability of the applicable codecs. Given a character set, it will do its best to provide information on how to use that character set in an email message in an RFC-compliant way.

Certain character sets must be encoded with quoted-printable or base64 when used in email headers or bodies. Certain character sets must be converted outright, and are not allowed in email.

Optional `input_charset` is as described below; it is always coerced to lower case. After being alias normalized it is also used as a lookup into the registry of character sets to find out the header encoding, body encoding, and output conversion codec to be used for the character set. For example, if `input_charset` is `iso-8859-1`, then headers and bodies will be encoded using
quoted-printable and no output conversion codec is necessary. If input_charset is euc-jp, then headers will be encoded with base64, bodies will not be encoded, but output text will be converted from the euc-jp character set to the iso-2022-jp character set.

Charset instances have the following data attributes:

**input_charset**

The initial character set specified. Common aliases are converted to their official email names (e.g. latin_1 is converted to iso-8859-1). Defaults to 7-bit us-ascii.

**header_encoding**

If the character set must be encoded before it can be used in an email header, this attribute will be set to Charset.QP (for quoted-printable), Charset.BASE64 (for base64 encoding), or Charset.SHORTEST for the shortest of QP or BASE64 encoding. Otherwise, it will be None.

**body_encoding**

Same as header_encoding, but describes the encoding for the mail message's body, which indeed may be different than the header encoding. Charset.SHORTEST is not allowed for body_encoding.

**output_charset**

Some character sets must be converted before they can be used in email headers or bodies. If the input_charset is one of them, this attribute will contain the name of the character set output will be converted to. Otherwise, it will be None.

**input_codec**

The name of the Python codec used to convert the input_charset to Unicode. If no conversion codec is
necessary, this attribute will be `None`.

**output_codec**

The name of the Python codec used to convert Unicode to the `output_charset`. If no conversion codec is necessary, this attribute will have the same value as the `input_codec`.

**Charset** instances also have the following methods:

**get_body_encoding()**

Return the content transfer encoding used for body encoding.

This is either the string `quoted-printable` or `base64` depending on the encoding used, or it is a function, in which case you should call the function with a single argument, the Message object being encoded. The function should then set the `Content-Transfer-Encoding` header itself to whatever is appropriate.

Returns the string `quoted-printable` if `body_encoding` is `QP`, returns the string `base64` if `body_encoding` is `BASE64`, and returns the string `7bit` otherwise.

**convert(s)**

Convert the string `s` from the `input_codec` to the `output_codec`.

**to_splittable(s)**

Convert a possibly multibyte string to a safely splittable format. `s` is the string to split.

Uses the `input_codec` to try and convert the string to Unicode, so it can be safely split on character boundaries (even for multibyte characters).
Returns the string as-is if it isn’t known how to convert s to Unicode with the `input_charset`.

Characters that could not be converted to Unicode will be replaced with the Unicode replacement character 'U+FFFD'.

`from_splittable(ustr[, to_output])`
Convert a splittable string back into an encoded string. `ustr` is a Unicode string to “unsplit”.

This method uses the proper codec to try and convert the string from Unicode back into an encoded format. Return the string as-is if it is not Unicode, or if it could not be converted from Unicode.

Characters that could not be converted from Unicode will be replaced with an appropriate character (usually '?').

If `to_output` is `True` (the default), uses `output_codec` to convert to an encoded format. If `to_output` is `False`, it uses `input_codec`.

`get_output_charset()`
Return the output character set.

This is the `output_charset` attribute if that is not `None`, otherwise it is `input_charset`.

`encoded_header_len()`
Return the length of the encoded header string, properly calculating for quoted-printable or base64 encoding.

`header_encode(s[, convert])`
Header-encode the string `s`. 
If `convert` is `True`, the string will be converted from the input charset to the output charset automatically. This is not useful for multibyte character sets, which have line length issues (multibyte characters must be split on a character, not a byte boundary); use the higher-level `Header` class to deal with these issues (see `email.header`). `convert` defaults to `False`.

The type of encoding (base64 or quoted-printable) will be based on the `header_encoding` attribute.

**body_encode**(s[, `convert`])

Body-encode the string s.

If `convert` is `True` (the default), the string will be converted from the input charset to output charset automatically. Unlike `header_encode()`, there are no issues with byte boundaries and multibyte charsets in email bodies, so this is usually pretty safe.

The type of encoding (base64 or quoted-printable) will be based on the `body_encoding` attribute.

The `Charset` class also provides a number of methods to support standard operations and built-in functions.

```
__str__()  
Returns `input_charset` as a string coerced to lower case.  __repr__() is an alias for __str__().

__eq__(other)  
This method allows you to compare two `Charset` instances for equality.

__ne__(other)
```
This method allows you to compare two Charset instances for inequality.

The email.charset module also provides the following functions for adding new entries to the global character set, alias, and codec registries:

email.charset.add_charset(charset[, header_enc[, body_enc[, output_charset]]]])

Add character properties to the global registry.

charset is the input character set, and must be the canonical name of a character set.

Optional header_enc and body_enc is either Charset.QP for quoted-printable, Charset.BASE64 for base64 encoding, Charset.SHORTEST for the shortest of quoted-printable or base64 encoding, or None for no encoding. SHORTEST is only valid for header_enc. The default is None for no encoding.

Optional output_charset is the character set that the output should be in. Conversions will proceed from input charset, to Unicode, to the output charset when the method Charset.convert() is called. The default is to output in the same character set as the input.

Both input_charset and output_charset must have Unicode codec entries in the module’s character set-to-codec mapping; use add_codec() to add codecs the module does not know about. See the codecs module’s documentation for more information.

The global character set registry is kept in the module global dictionary CHARSETS.
email.charset.add_alias(alias, canonical)

Add a character set alias. alias is the alias name, e.g. latin-1. canonical is the character set’s canonical name, e.g. iso-8859-1.

The global charset alias registry is kept in the module global dictionary ALIASES.

eemail.charset.add_codec(charset, codecname)

Add a codec that map characters in the given character set to and from Unicode.

charset is the canonical name of a character set. codecname is the name of a Python codec, as appropriate for the second argument to the unicode() built-in, or to the encode() method of a Unicode string.
18.1. email — An email and MIME handling package
18.1.7. `email.encoders`: Encoders

When creating `Message` objects from scratch, you often need to encode the payloads for transport through compliant mail servers. This is especially true for `image/*` and `text/*` type messages containing binary data.

The `email` package provides some convenient encodings in its `encoders` module. These encoders are actually used by the `MIMEAudio` and `MIMEImage` class constructors to provide default encodings. All encoder functions take exactly one argument, the message object to encode. They usually extract the payload, encode it, and reset the payload to this newly encoded value. They should also set the `Content-Transfer-Encoding` header as appropriate.

Note that these functions are not meaningful for a multipart message. They must be applied to individual subparts instead, and will raise a `TypeError` if passed a message whose type is multipart.

Here are the encoding functions provided:

```python
e-mail.encoders.encode_quopri(msg)
# Encodes the payload into quoted-printable form and sets the
# Content-Transfer-Encoding header to quoted-printable [1]. This
# is a good encoding to use when most of your payload is normal
# printable data, but contains a few unprintable characters.
```

```python
e-mail.encoders.encode_base64(msg)
# Encodes the payload into base64 form and sets the Content-
# Transfer-Encoding header to base64. This is a good encoding to
# use when most of your payload is unprintable data since it is a
# more compact form than quoted-printable. The drawback of
# base64 encoding is that it renders the text non-human readable.
```
**email.encoders.** `encode_7or8bit(msg)`

This doesn’t actually modify the message’s payload, but it does set the `Content-Transfer-Encoding` header to either 7bit or 8bit as appropriate, based on the payload data.

**email.encoders.** `encode_noop(msg)`

This does nothing; it doesn’t even set the `Content-Transfer-Encoding` header.

**Footnotes**

[1] Note that encoding with `encode_quopri()` also encodes all tabs and space characters in the data.
18.1.8. **email.errors**: Exception and Defect classes

The following exception classes are defined in the `email.errors` module:

**exception** `email.errors.MessageError`

This is the base class for all exceptions that the `email` package can raise. It is derived from the standard `Exception` class and defines no additional methods.

**exception** `email.errors.MessageParseError`

This is the base class for exceptions raised by the `Parser` class. It is derived from `MessageError`.

**exception** `email.errors.HeaderParseError`

Raised under some error conditions when parsing the `RFC 2822` headers of a message, this class is derived from `MessageParseError`. It can be raised from the `Parser.parse` or `Parser.parsestr` methods.

Situations where it can be raised include finding an envelope header after the first `RFC 2822` header of the message, finding a continuation line before the first `RFC 2822` header is found, or finding a line in the headers which is neither a header or a continuation line.

**exception** `email.errors.BoundaryError`

Raised under some error conditions when parsing the `RFC 2822` headers of a message, this class is derived from `MessageParseError`. It can be raised from the `Parser.parse` or `Parser.parsestr` methods.
Situations where it can be raised include not being able to find the starting or terminating boundary in a `multipart/*` message when strict parsing is used.

**exception** `email.errors.MultipartConversionError`

Raised when a payload is added to a `Message` object using `add_payload()`, but the payload is already a scalar and the message's `Content-Type` main type is not either `multipart` or missing. `MultipartConversionError` multiply inherits from `MessageError` and the built-in `TypeError`.

Since `Message.add_payload()` is deprecated, this exception is rarely raised in practice. However the exception may also be raised if the `attach()` method is called on an instance of a class derived from `MIMENonMultipart` (e.g. `MIMEImage`).

Here's the list of the defects that the `FeedParser` can find while parsing messages. Note that the defects are added to the message where the problem was found, so for example, if a message nested inside a `multipart/alternative` had a malformed header, that nested message object would have a defect, but the containing messages would not.

All defect classes are subclassed from `email.errors.MessageDefect`, but this class is *not* an exception!

**New in version 2.4:** All the defect classes were added.

- `NoBoundaryInMultipartDefect` – A message claimed to be a multipart, but had no `boundary` parameter.
- `StartBoundaryNotFoundDefect` – The start boundary claimed in the `Content-Type` header was never found.
- `FirstHeaderLineIsContinuationDefect` – The message had a continuation line as its first header line.
- **MisplacedEnvelopeHeaderDefect** - A “Unix From” header was found in the middle of a header block.
- **MalformedHeaderDefect** – A header was found that was missing a colon, or was otherwise malformed.
- **MultipartInvariantViolationDefect** – A message claimed to be a *multipart*, but no subparts were found. Note that when a message has this defect, its `is_multipart()` method may return false even though its content type claims to be *multipart*. 
18.1.9. email.utils: Miscellaneous utilities

There are several useful utilities provided in the `email.utils` module:

`email.utils.quote(str)`
Return a new string with backslashes in `str` replaced by two backslashes, and double quotes replaced by backslash-double quote.

`email.utils.unquote(str)`
Return a new string which is an unquoted version of `str`. If `str` ends and begins with double quotes, they are stripped off. Likewise if `str` ends and begins with angle brackets, they are stripped off.

`email.utils.parseaddr(address)`
Parse address – which should be the value of some address-containing field such as `To` or `Cc` – into its constituent `realname` and `email address` parts. Returns a tuple of that information, unless the parse fails, in which case a 2-tuple of (',', '') is returned.

`email.utils.formataddr(pair)`
The inverse of `parseaddr()`, this takes a 2-tuple of the form (realname, email_address) and returns the string value suitable for a `To` or `Cc` header. If the first element of `pair` is false, then the second element is returned unmodified.

`email.utils.getaddresses(fieldvalues)`
This method returns a list of 2-tuples of the form returned by `parseaddr()`. `fieldvalues` is a sequence of header field values as might be returned by `Message.get_all`. Here’s a simple example
that gets all the recipients of a message:

```python
from email.utils import getaddresses
tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

e-mail.utils. **parsedate**(date)

Attempts to parse a date according to the rules in [RFC 2822](https://tools.ietf.org/html/rfc2822). However, some mailers don’t follow that format as specified, so `parsedate()` tries to guess correctly in such cases. `date` is a string containing an [RFC 2822](https://tools.ietf.org/html/rfc2822) date, such as "Mon, 20 Nov 1995 19:12:08 -0500". If it succeeds in parsing the date, `parsedate()` returns a 9-tuple that can be passed directly to `time.mktime()`; otherwise `None` will be returned. Note that indexes 6, 7, and 8 of the result tuple are not usable.

e-mail.utils. **parsedate_tz**(date)

Performs the same function as `parsedate()`, but returns either `None` or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to `time.mktime()`, and the tenth is the offset of the date’s timezone from UTC (which is the official term for Greenwich Mean Time) [1]. If the input string has no timezone, the last element of the tuple returned is `None`. Note that indexes 6, 7, and 8 of the result tuple are not usable.

e-mail.utils. **mktime_tz**(tuple)

Turn a 10-tuple as returned by `parsedate_tz()` into a UTC timestamp (seconds since the Epoch). If the timezone item in the tuple is `None`, assume local time.

e-mail.utils. **formatdate**(tuple)

Attempts to express the [parsed](https://tools.ietf.org/html/rfc2822) date as a string; if successful, it will return an 8-tuple corresponding to a string of the form `Mon, 20 Nov 1995 19:12:08 -0500`. Otherwise, `None` is returned.
Returns a date string as per RFC 2822, e.g.:

```
Fri, 09 Nov 2001 01:08:47 -0000
```

Optional `timeval` if given is a floating point time value as accepted by `time.gmtime()` and `time.localtime()`, otherwise the current time is used.

Optional `localtime` is a flag that when `True`, interprets `timeval`, and returns a date relative to the local timezone instead of UTC, properly taking daylight savings time into account. The default is `False` meaning UTC is used.

Optional `usegmt` is a flag that when `True`, outputs a date string with the timezone as an ascii string `GMT`, rather than a numeric `-0000`. This is needed for some protocols (such as HTTP). This only applies when `localtime` is `False`. The default is `False`.

*New in version 2.4.*

```python
email.utils.make_msgid([idstring])
```

Returns a string suitable for an RFC 2822-compliant Message-ID header. Optional `idstring` if given, is a string used to strengthen the uniqueness of the message id.

```python
email.utils.decode_rfc2231(s)
```

Decode the string `s` according to RFC 2231.

```python
email.utils.encode_rfc2231(s[, charset[, language]])
```

Encode the string `s` according to RFC 2231. Optional `charset` and `language`, if given is the character set name and language name to use. If neither is given, `s` is returned as-is. If `charset` is given but `language` is not, the string is encoded using the empty string for `language`.
email.utils.**collapse_rfc2231_value**(value[, errors[, fallback_charset]])

When a header parameter is encoded in **RFC 2231** format, `Message.get_param` may return a 3-tuple containing the character set, language, and value. `collapse_rfc2231_value()` turns this into a unicode string. Optional `errors` is passed to the `errors` argument of the built-in `unicode()` function; it defaults to `replace`. Optional `fallback_charset` specifies the character set to use if the one in the **RFC 2231** header is not known by Python; it defaults to `us-ascii`.

For convenience, if the `value` passed to `collapse_rfc2231_value()` is not a tuple, it should be a string and it is returned unquoted.

email.utils.**decode_params**(params)

Decode parameters list according to **RFC 2231**. `params` is a sequence of 2-tuples containing elements of the form `(content-type, string-value)`.

*Changed in version 2.4:* The `dump_address_pair()` function has been removed; use `formataddr()` instead.

*Changed in version 2.4:* The `decode()` function has been removed; use the `Header.decode_header` method instead.

*Changed in version 2.4:* The `encode()` function has been removed; use the `Header.encode` method instead.

**Footnotes**

Note that the sign of the timezone offset is the opposite of the sign of the `time.timezone` variable for the same timezone; the
latter variable follows the POSIX standard while this module follows RFC 2822.
18.1. **email** — An email and MIME handling package
Iterating over a message object tree is fairly easy with the `Message.walk` method. The `email.iterators` module provides some useful higher level iterations over message object trees.

**email.iterators.**`body_line_iterator`(msg[, decode])

This iterates over all the payloads in all the subparts of `msg`, returning the string payloads line-by-line. It skips over all the subpart headers, and it skips over any subpart with a payload that isn’t a Python string. This is somewhat equivalent to reading the flat text representation of the message from a file using `readline()`, skipping over all the intervening headers.

Optional `decode` is passed through to `Message.get_payload`.

**email.iterators.**`typed_subpart_iterator`(msg[, maintype[, subtype]])

This iterates over all the subparts of `msg`, returning only those subparts that match the MIME type specified by `maintype` and `subtype`.

Note that `subtype` is optional; if omitted, then subpart MIME type matching is done only with the main type. `maintype` is optional too; it defaults to `text`.

Thus, by default `typed_subpart_iterator()` returns each subpart that has a MIME type of `text/*`.

The following function has been added as a useful debugging tool. It should not be considered part of the supported public interface for the package.
email.iterators._structure(msg[, fp[, level]])

Prints an indented representation of the content types of the message object structure. For example:

```python
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
text/plain
text/plain
multipart/digest
  message/rfc822
text/plain
  message/rfc822
text/plain
  message/rfc822
text/plain
  message/rfc822
text/plain
  message/rfc822
text/plain
text/plain
```

Optional `fp` is a file-like object to print the output to. It must be suitable for Python’s extended print statement. `level` is used internally.
18.1.11. email: Examples

Here are a few examples of how to use the email package to read, write, and send simple email messages, as well as more complex MIME messages.

First, let's see how to create and send a simple text message:

```python
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.mime.text import MIMEText

# Open a plain text file for reading. For this example, assume
# the text file contains only ASCII characters.
fp = open(textfile, 'rb')

# Create a text/plain message
msg = MIMEText(fp.read())
fp.close()

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = 'The contents of %s' % textfile
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server, but don't include the
# envelope header.
s = smtplib.SMTP('localhost')
s.sendmail(me, [you], msg.as_string())
s.quit()

And parsing RFC822 headers can easily be done by the
parse(filename) or parsestr(message_as_string) methods of the
Parser() class:

```
# If the e-mail headers are in a file, uncomment this line:
#headers = Parser().parse(open(messagefile, 'r'))

# Or for parsing headers in a string, use:
headers = Parser().parsestr('From: <user@example.com>
  'To: <someone_else@example.com>
  'Subject: Test message
  
  'Body would go here
)

# Now the header items can be accessed as a dictionary:
print 'To: %s' % headers['to']
print 'From: %s' % headers['from']
print 'Subject: %s' % headers['subject']

Here's an example of how to send a MIME message containing a bunch of family pictures that may be residing in a directory:

# Import smtplib for the actual sending function
import smtplib

# Here are the email package modules we'll need
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart

COMMASPACE = ',', '

# Create the container (outer) email message.
msg = MIMEMultipart()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = COMMASPACE.join(family)
msg.preamble = 'Our family reunion'

# Assume we know that the image files are all in PNG format
for file in pngfiles:
  # Open the files in binary mode. Let the MIMEImage class a
guess the specific image type.
  fp = open(file, 'rb')
  img = MIMEImage(fp.read())
  fp.close()
  msg.attach(img)
Here’s an example of how to send the entire contents of a directory as an email message: [1]

```python
#!/usr/bin/env python

"""Send the contents of a directory as a MIME message.""

import os
import sys
import smtplib
import mimetypes
from optparse import OptionParser
from email import encoders
from email.message import Message
from email.mime.audio import MIMEAudio
from email.mime.base import MIMEBase
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

COMMASPACE = ', '

def main():
    parser = OptionParser(usage="""
Send the contents of a directory as a MIME message.

Usage: %prog [options]
"""
   Unless the -o option is given, the email is sent by forwarding
   SMTP server, which then does the normal delivery process. Your
   must be running an SMTP server.

    parser.add_option('-d', '--directory',
                      type='string', action='store',
                      help="""Mail the contents of the specified directory.
otherwise use the current directory. Onl"""
```
```
files in the directory are sent, and we don't recurse to subdirectories.

```python
def files_in_directory(directory):
    files = os.listdir(directory)
    for filename in files:
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        ctype, encoding = mimetypes.guess_type(path)
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded (e.g., gzip'd or compressed)
            ctype = 'application/octet-stream'
            maintype, subtype = ctype.split('/', 1)
        if maintype == 'text':
            fp = open(path, 'r')
            msg = MIMEText(fp.read(), _subtype=subtype)
            fp.close()
        elif maintype == 'image':
            fp = open(path, 'rb')
```

msg = MIMEImage(fp.read(), _subtype=subtype)
fp.close()

elif maintype == 'audio':
    fp = open(path, 'rb')
    msg = MIMEAudio(fp.read(), _subtype=subtype)
    fp.close()
else:
    fp = open(path, 'rb')
    msg = MIMEBase(maintype, subtype)
    msg.set_payload(fp.read())
    fp.close()
    # Encode the payload using Base64
    encoders.encode_base64(msg)
    # Set the filename parameter
    msg.add_header('Content-Disposition', 'attachment', filename=outer)
outer.attach(msg)

# Now send or store the message
composed = outer.as_string()

if opts.output:
    fp = open(opts.output, 'w')
    fp.write(composed)
    fp.close()
else:
    s = smtplib.SMTP('localhost')
s.sendmail(opts.sender, opts.recipients, composed)
s.quit()

if __name__ == '__main__':
    main()
def main():
    parser = OptionParser(usage="Unpack a MIME message into a directory of files.

Usage: %prog [options] msgfile
"")
    parser.add_option('-d', '--directory',
                      type='string', action='store',
                      help="Unpack the MIME message into the directory, which will be created if it does not already exist.")

    opts, args = parser.parse_args()
    if not opts.directory:
        parser.print_help()
        sys.exit(1)

    try:
        msgfile = args[0]
    except IndexError:
        parser.print_help()
        sys.exit(1)

    try:
        os.mkdir(opts.directory)
    except OSError as e:
        # Ignore directory exists error
        if e.errno != errno.EEXIST:
            raise

    fp = open(msgfile)
    msg = email.message_from_file(fp)
    fp.close()

    counter = 1
    for part in msg.walk():
        # multipart/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given filename
        # email message can't be used to overwrite important files
        filename = part.get_filename()
        if not filename:
            ext = mimetypes.guess_extension(part.get_content_type())
            if not ext:
                # Use a generic bag-of-bits extension
Here's an example of how to create an HTML message with an alternative plain text version: [2]

```python
#!/usr/bin/env python

import smtplib

from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

# me == my email address
# you == recipient's email address
me = "my@email.com"
you = "your@email.com"

# Create message container - the correct MIME type is multipart.
msg = MIMEMultipart('alternative')
msg['Subject'] = "Link"
msg['From'] = me
msg['To'] = you

text = "Hi!
How are you?
Here is the link you wanted:
https://www.python.org"

html = "
<html>
<head>
</head>
<body>
<p>Hi!</p>
  How are you?
  <br>
  Here is the <a href="https://www.python.org">link</a> you
</p>
</body>
</html>
"
```

Record the MIME types of both parts - text/plain and text/html.
part1 = MIMEText(text, 'plain')
part2 = MIMEText(html, 'html')

Attach parts into message container.
According to RFC 2046, the last part of a multipart message, the HTML message, is best and preferred.
msg.attach(part1)
msg.attach(part2)

Send the message via local SMTP server.
s = smtplib.SMTP('localhost')
sendmail function takes 3 arguments: sender's address, recipient's address and message to send - here it is sent as one string.
s.sendmail(me, you, msg.as_string())
s.quit()

Footnotes

[1] Thanks to Matthew Dixon Cowles for the original inspiration and examples.
18.2. json — JSON encoder and decoder

New in version 2.6.

JSON (JavaScript Object Notation), specified by RFC 7159 (which obsoletes RFC 4627) and by ECMA-404, is a lightweight data interchange format inspired by JavaScript object literal syntax (although it is not a strict subset of JavaScript [1]).

json exposes an API familiar to users of the standard library marshal and pickle modules.

Encoding basic Python object hierarchies:

```python
>>> import json
>>> json.dumps([{'foo', {'bar': ('baz', None, 1.0, 2)}]})
'"foo", {"bar": ["baz", null, 1.0, 2]}"
>>> print json.dumps("\foo\bar")
"\foo\bar"
>>> print json.dumps(u'\u1234')
\u1234
>>> print json.dumps('\')
\"
>>> print json.dumps({'c': 0, "b": 0, 'a': 0}, sort_keys=True)
{"a": 0, "b": 0, "c": 0}
>>> from StringIO import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'"streaming API"'

Compact encoding:

```python
>>> import json
>>> json.dumps([1,2,3,{'4': 5, '6': 7}], separators=(',',':'))
'[1,2,3,{"4":5,"6":7}]'
```
Pretty printing:
>>> import json
>>> print json.dumps({'4': 5, '6': 7}, sort_keys=True,
...
indent=4, separators=(',', ': '))
{
"4": 5,
"6": 7
}

Decoding JSON:
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
[u'foo', {u'bar': [u'baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
u'"foo\x08ar'
>>> from StringIO import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
[u'streaming API']

Specializing JSON object decoding:
>>> import json
>>> def as_complex(dct):
...
if '__complex__' in dct:
...
return complex(dct['real'], dct['imag'])
...
return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...
object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')

Extending

JSONEncoder :

>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...
def default(self, obj):
...
if isinstance(obj, complex):


...  return [obj.real, obj.imag]
...  # Let the base class default method raise the TypeError
...  return json.JSONEncoder.default(self, obj)

>>> dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[', '2.0', ',', ',', '1.0', ',', ']']

Using `json.tool` from the shell to validate and pretty-print:

```bash
$ echo '{"json":"obj"}' | python -m json.tool
{
   "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

**Note:** JSON is a subset of YAML 1.2. The JSON produced by this module's default settings (in particular, the default `separators` value) is also a subset of YAML 1.0 and 1.1. This module can thus also be used as a YAML serializer.
json.dump(obj, fp, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, encoding="utf-8", default=None, sort_keys=False, **kw)

Serialize obj as a JSON formatted stream to fp (a write-supporting file-like object) using this conversion table.

If skipkeys is true (default: False), then dict keys that are not of a basic type (str, unicode, int, long, float, bool, None) will be skipped instead of raising a TypeError.

If ensure_ascii is true (the default), all non-ASCII characters in the output are escaped with \uxxxx sequences, and the result is a str instance consisting of ASCII characters only. If ensure_ascii is false, some chunks written to fp may be unicode instances. This usually happens because the input contains unicode strings or the encoding parameter is used. Unless fp.write() explicitly understands unicode (as in codecs.getwriter()) this is likely to cause an error.

If check_circular is false (default: True), then the circular reference check for container types will be skipped and a circular reference will result in an OverflowError (or worse).

If allow_nan is false (default: True), then it will be a ValueError to serialize out of range float values (nan, inf, -inf) in strict compliance of the JSON specification. If allow_nan is true, their JavaScript equivalents (NaN, Infinity, -Infinity) will be used.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An
indent level of 0, or negative, will only insert newlines. None (the default) selects the most compact representation.

**Note:** Since the default item separator is ',', the output might include trailing whitespace when *indent* is specified. You can use *separators=('', ' : ')* to avoid this.

If specified, *separators* should be an *(item_separator, key_separator)* tuple. By default, ('', ' : ') are used. To get the most compact JSON representation, you should specify ('', ': ') to eliminate whitespace.

*encoding* is the character encoding for *str* instances, default is UTF-8.

If specified, *default* should be a function that gets called for objects that can’t otherwise be serialized. It should return a JSON encodable version of the object or raise a *TypeError*. If not specified, *TypeError* is raised.

If *sort_keys* is true (default: *False*), then the output of dictionaries will be sorted by key.

To use a custom *JSONEncoder* subclass (e.g. one that overrides the *default()* method to serialize additional types), specify it with the *cls* kwarg; otherwise *JSONEncoder* is used.

**Note:** Unlike *pickle* and *marshal*, JSON is not a framed protocol so trying to serialize more objects with repeated calls to *dump()* and the same *fp* will result in an invalid JSON file.

```python
json.dumps(obj, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, encoding="utf-8", default=None, sort_keys=False,
```
**kw)

Serialize obj to a JSON formatted str using this conversion table. If ensure_ascii is false, the result may contain non-ASCII characters and the return value may be a unicode instance.

The arguments have the same meaning as in dump().

**Note:** Keys in key/value pairs of JSON are always of the type str. When a dictionary is converted into JSON, all the keys of the dictionary are coerced to strings. As a result of this, if a dictionary is converted into JSON and then back into a dictionary, the dictionary may not equal the original one. That is, loads(dumps(x)) ≠ x if x has non-string keys.

json.load(fp[, encoding[, cls[, object_hook[, parse_float[, parse_int[, parse_constant[, object_pairs_hook[, **kw]]]]]]]])

Deserialize fp (a read()-supporting file-like object containing a JSON document) to a Python object using this conversion table.

If the contents of fp are encoded with an ASCII based encoding other than UTF-8 (e.g. latin-1), then an appropriate encoding name must be specified. Encodings that are not ASCII based (such as UCS-2) are not allowed, and should be wrapped with codecs.getreader(encoding)(fp), or simply decoded to a unicode object and passed to loads().

object_hook is an optional function that will be called with the result of any object literal decoded (a dict). The return value of object_hook will be used instead of the dict. This feature can be used to implement custom decoders (e.g. JSON-RPC class hinting).

object_pairs_hook is an optional function that will be called with
the result of any object literal decoded with an ordered list of pairs. The return value of `object_pairs_hook` will be used instead of the `dict`. This feature can be used to implement custom decoders that rely on the order that the key and value pairs are decoded (for example, `collections.OrderedDict()` will remember the order of insertion). If `object_hook` is also defined, the `object_pairs_hook` takes priority.

*Changed in version 2.7:* Added support for `object_pairs_hook`.

`parse_float`, if specified, will be called with the string of every JSON float to be decoded. By default, this is equivalent to `float(num_str)`. This can be used to use another datatype or parser for JSON floats (e.g. `decimal.Decimal`).

`parse_int`, if specified, will be called with the string of every JSON int to be decoded. By default, this is equivalent to `int(num_str)`. This can be used to use another datatype or parser for JSON integers (e.g. `float`).

`parse_constant`, if specified, will be called with one of the following strings: `-Infinity`, `'Infinity'`, `'NaN'`. This can be used to raise an exception if invalid JSON numbers are encountered.

*Changed in version 2.7:* `parse_constant` doesn’t get called on ‘null’, ‘true’, ‘false’ anymore.

To use a custom `JSONDecoder` subclass, specify it with the `cls` kwarg; otherwise `JSONDecoder` is used. Additional keyword arguments will be passed to the constructor of the class.

```python
json.loads(s[, encoding[, cls[, object_hook[, parse_float[, parse_int[, parse_constant[, object_pairs_hook[, **kw]]]]]]]])
```
Deserialize s (a `str` or `unicode` instance containing a JSON document) to a Python object using this conversion table.

If s is a `str` instance and is encoded with an ASCII based encoding other than UTF-8 (e.g. latin-1), then an appropriate `encoding` name must be specified. Encodings that are not ASCII based (such as UCS-2) are not allowed and should be decoded to `unicode` first.

The other arguments have the same meaning as in `load()`.
18.2.2. Encoders and Decoders

```python
class json.JSONDecoder([encoding[, object_hook[, parse_float[, parse_int[, parse_constant[, strict[, object_pairs_hook]]]]]]])
```

Simple JSON decoder.

Performs the following translations in decoding by default:

<table>
<thead>
<tr>
<th>JSON</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>dict</td>
</tr>
<tr>
<td>array</td>
<td>list</td>
</tr>
<tr>
<td>string</td>
<td>unicode</td>
</tr>
<tr>
<td>number (int)</td>
<td>int, long</td>
</tr>
<tr>
<td>number (real)</td>
<td>float</td>
</tr>
<tr>
<td>true</td>
<td>True</td>
</tr>
<tr>
<td>false</td>
<td>False</td>
</tr>
<tr>
<td>null</td>
<td>None</td>
</tr>
</tbody>
</table>

It also understands NaN, Infinity, and -Infinity as their corresponding float values, which is outside the JSON spec.

`encoding` determines the encoding used to interpret any str objects decoded by this instance (UTF-8 by default). It has no effect when decoding unicode objects.

Note that currently only encodings that are a superset of ASCII work, strings of other encodings should be passed in as unicode.

`object_hook`, if specified, will be called with the result of every JSON object decoded and its return value will be used in place of the given dict. This can be used to provide custom deserializations (e.g. to support JSON-RPC class hinting).
object_pairs_hook, if specified will be called with the result of every JSON object decoded with an ordered list of pairs. The return value of object_pairs_hook will be used instead of the dict. This feature can be used to implement custom decoders that rely on the order that the key and value pairs are decoded (for example, collections.OrderedDict() will remember the order of insertion). If object_hook is also defined, the object_pairs_hook takes priority.

Changed in version 2.7: Added support for object_pairs_hook.

parse_float, if specified, will be called with the string of every JSON float to be decoded. By default, this is equivalent to float(num_str). This can be used to use another datatype or parser for JSON floats (e.g. decimal.Decimal).

parse_int, if specified, will be called with the string of every JSON int to be decoded. By default, this is equivalent to int(num_str). This can be used to use another datatype or parser for JSON integers (e.g. float).

parse_constant, if specified, will be called with one of the following strings: '-Infinity', 'Infinity', 'NaN'. This can be used to raise an exception if invalid JSON numbers are encountered.

If strict is false (True is the default), then control characters will be allowed inside strings. Control characters in this context are those with character codes in the 0–31 range, including \t (tab), \n, \r and \0.

If the data being deserialized is not a valid JSON document, a ValueError will be raised.
decode(s)
Return the Python representation of s (a str or unicode instance containing a JSON document).

raw_decode(s)
Decode a JSON document from s (a str or unicode beginning with a JSON document) and return a 2-tuple of the Python representation and the index in s where the document ended.

This can be used to decode a JSON document from a string that may have extraneous data at the end.

class json.JSONEncoder([skipkeys[, ensure_ascii[, check_circular[, allow_nan[, sort_keys[, indent[, separators[, encoding[, default]]]]]]]]]
Extensible JSON encoder for Python data structures.

Supports the following objects and types by default:

<table>
<thead>
<tr>
<th>Python</th>
<th>JSON</th>
</tr>
</thead>
<tbody>
<tr>
<td>dict</td>
<td>object</td>
</tr>
<tr>
<td>list, tuple</td>
<td>array</td>
</tr>
<tr>
<td>str, unicode</td>
<td>string</td>
</tr>
<tr>
<td>int, long, float</td>
<td>number</td>
</tr>
<tr>
<td>True</td>
<td>true</td>
</tr>
<tr>
<td>False</td>
<td>false</td>
</tr>
<tr>
<td>None</td>
<td>null</td>
</tr>
</tbody>
</table>

To extend this to recognize other objects, subclass and implement a default() method with another method that returns a serializable object for o if possible, otherwise it should call the superclass implementation (to raise TypeError).

If skipkeys is false (the default), then it is a TypeError to attempt
encoding of keys that are not str, int, long, float or None. If skipkeys is true, such items are simply skipped.

If ensure_ascii is true (the default), all non-ASCII characters in the output are escaped with \uXXXX sequences, and the results are str instances consisting of ASCII characters only. If ensure_ascii is false, a result may be a unicode instance. This usually happens if the input contains unicode strings or the encoding parameter is used.

If check_circular is true (the default), then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true (the default), then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true (default: False), then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer (it is None by default), then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

**Note:** Since the default item separator is ",", the output might include trailing whitespace when indent is specified. You can use separators=(',', ' : ') to avoid this.
If specified, *separators* should be an `(item_separator, key_separator)` tuple. By default, `(',', ':')` are used. To get the most compact JSON representation, you should specify `(', ', ':')` to eliminate whitespace.

If specified, *default* should be a function that gets called for objects that can’t otherwise be serialized. It should return a JSON encodable version of the object or raise a *TypeError*. If not specified, *TypeError* is raised.

If *encoding* is not *None*, then all input strings will be transformed into unicode using that encoding prior to JSON-encoding. The default is UTF-8.

**default**(o)

Implement this method in a subclass such that it returns a serializable object for *o*, or calls the base implementation (to raise a *TypeError*).

For example, to support arbitrary iterators, you could implement `default` like this:

```python
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

**encode**(o)

Return a JSON string representation of a Python data structure, *o*. For example:
iterencode(o)

Encode the given object, o, and yield each string representation as available. For example:

```python
for chunk in JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```
18.2.3. Standard Compliance and Interoperability

The JSON format is specified by RFC 7159 and by ECMA-404. This section details this module’s level of compliance with the RFC. For simplicity, JSONEncoder and JSONDecoder subclasses, and parameters other than those explicitly mentioned, are not considered.

This module does not comply with the RFC in a strict fashion, implementing some extensions that are valid JavaScript but not valid JSON. In particular:

- Infinite and NaN number values are accepted and output;
- Repeated names within an object are accepted, and only the value of the last name-value pair is used.

Since the RFC permits RFC-compliant parsers to accept input texts that are not RFC-compliant, this module’s deserializer is technically RFC-compliant under default settings.

18.2.3.1. Character Encodings

The RFC requires that JSON be represented using either UTF-8, UTF-16, or UTF-32, with UTF-8 being the recommended default for maximum interoperability. Accordingly, this module uses UTF-8 as the default for its encoding parameter.

This module’s deserializer only directly works with ASCII-compatible encodings; UTF-16, UTF-32, and other ASCII-incompatible encodings require the use of workarounds described in the documentation for the deserializer’s encoding parameter.

As permitted, though not required, by the RFC, this module’s
serializer sets `ensure_ascii=True` by default, thus escaping the output so that the resulting strings only contain ASCII characters.

The RFC prohibits adding a byte order mark (BOM) to the start of a JSON text, and this module’s serializer does not add a BOM to its output. The RFC permits, but does not require, JSON deserializers to ignore an initial BOM in their input. This module’s deserializer raises a `ValueError` when an initial BOM is present.

The RFC does not explicitly forbid JSON strings which contain byte sequences that don’t correspond to valid Unicode characters (e.g. unpaired UTF-16 surrogates), but it does note that they may cause interoperability problems. By default, this module accepts and outputs (when present in the original `str`) code points for such sequences.

### 18.2.3.2. Infinite and NaN Number Values

The RFC does not permit the representation of infinite or NaN number values. Despite that, by default, this module accepts and outputs `Infinity`, `-Infinity`, and `NaN` as if they were valid JSON number literal values:

```python
>>> # Neither of these calls raises an exception, but the result
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

In the serializer, the `allow_nan` parameter can be used to alter this behavior. In the deserializer, the `parse_constant` parameter can be
used to alter this behavior.

18.2.3.3. Repeated Names Within an Object

The RFC specifies that the names within a JSON object should be unique, but does not mandate how repeated names in JSON objects should be handled. By default, this module does not raise an exception; instead, it ignores all but the last name-value pair for a given name:

```python
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{u'x': 3}
```

The `object_pairs_hook` parameter can be used to alter this behavior.

18.2.3.4. Top-level Non-Object, Non-Array Values

The old version of JSON specified by the obsolete RFC 4627 required that the top-level value of a JSON text must be either a JSON object or array (Python `dict` or `list`), and could not be a JSON null, boolean, number, or string value. RFC 7159 removed that restriction, and this module does not and has never implemented that restriction in either its serializer or its deserializer.

Regardless, for maximum interoperability, you may wish to voluntarily adhere to the restriction yourself.

18.2.3.5. Implementation Limitations

Some JSON deserializer implementations may set limits on:

- the size of accepted JSON texts
- the maximum level of nesting of JSON objects and arrays
- the range and precision of JSON numbers
- the content and maximum length of JSON strings

This module does not impose any such limits beyond those of the relevant Python datatypes themselves or the Python interpreter itself.

When serializing to JSON, beware any such limitations in applications that may consume your JSON. In particular, it is common for JSON numbers to be deserialized into IEEE 754 double precision numbers and thus subject to that representation's range and precision limitations. This is especially relevant when serializing Python `int` values of extremely large magnitude, or when serializing instances of “exotic” numerical types such as `decimal.Decimal`.

Footnotes

[1] As noted in the errata for RFC 7159, JSON permits literal U+2028 (LINE SEPARATOR) and U+2029 (PARAGRAPH SEPARATOR) characters in strings, whereas JavaScript (as of ECMAScript Edition 5.1) does not.
Mailcap files are used to configure how MIME-aware applications such as mail readers and Web browsers react to files with different MIME types. (The name “mailcap” is derived from the phrase “mail capability”.) For example, a mailcap file might contain a line like

```
video/mpeg; xmpeg %s
```

Then, if the user encounters an email message or Web document with the MIME type `video/mpeg`, `%s` will be replaced by a filename (usually one belonging to a temporary file) and the `xmpeg` program can be automatically started to view the file.

The mailcap format is documented in [RFC 1524](https://tools.ietf.org/html/rfc1524), “A User Agent Configuration Mechanism For Multimedia Mail Format Information,” but is not an Internet standard. However, mailcap files are supported on most Unix systems.

```python
mailcap.findmatch(caps, MIMEtype[, key[, filename[, plist]]])
```

Return a 2-tuple; the first element is a string containing the command line to be executed (which can be passed to `os.system()`), and the second element is the mailcap entry for a given MIME type. If no matching MIME type can be found, `(None, None)` is returned.

`key` is the name of the field desired, which represents the type of activity to be performed; the default value is ‘view’, since in the most common case you simply want to view the body of the MIME-typed data. Other possible values might be ‘compose’ and ‘edit’, if you wanted to create a new body of the given MIME type.
or alter the existing body data. See RFC 1524 for a complete list of these fields.

filename is the filename to be substituted for %s in the command line; the default value is '/dev/null' which is almost certainly not what you want, so usually you’ll override it by specifying a filename.

plist can be a list containing named parameters; the default value is simply an empty list. Each entry in the list must be a string containing the parameter name, an equals sign ('='), and the parameter’s value. Mailcap entries can contain named parameters like %{foo}, which will be replaced by the value of the parameter named 'foo'. For example, if the command line showpartial %{id} %{number} %{total} was in a mailcap file, and plist was set to ['id=1', 'number=2', 'total=3'], the resulting command line would be 'showpartial 1 2 3'.

In a mailcap file, the “test” field can optionally be specified to test some external condition (such as the machine architecture, or the window system in use) to determine whether or not the mailcap line applies. findmatch() will automatically check such conditions and skip the entry if the check fails.

mailcap.getcaps()

Returns a dictionary mapping MIME types to a list of mailcap file entries. This dictionary must be passed to the findmatch() function. An entry is stored as a list of dictionaries, but it shouldn’t be necessary to know the details of this representation.

The information is derived from all of the mailcap files found on the system. Settings in the user’s mailcap file $HOME/.mailcap will override settings in the system mailcap files /etc/mailcap, /usr/etc/mailcap, and /usr/local/etc/mailcap.
An example usage:

```python
>>> import mailcap
>>> d = mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='tmp1223')
('xmpeg tmp1223', {'view': 'xmpeg %s'})
```
This module defines two classes, `Mailbox` and `Message`, for accessing and manipulating on-disk mailboxes and the messages they contain. `Mailbox` offers a dictionary-like mapping from keys to messages. `Message` extends the `email.message` module's `Message` class with format-specific state and behavior. Supported mailbox formats are Maildir, mbox, MH, Babyl, and MMDF.

See also:

Module `email`

Represent and manipulate messages.
18.4.1. Mailbox objects

class mailbox.Mailbox

A mailbox, which may be inspected and modified.

The Mailbox class defines an interface and is not intended to be instantiated. Instead, format-specific subclasses should inherit from Mailbox and your code should instantiate a particular subclass.

The Mailbox interface is dictionary-like, with small keys corresponding to messages. Keys are issued by the Mailbox instance with which they will be used and are only meaningful to that Mailbox instance. A key continues to identify a message even if the corresponding message is modified, such as by replacing it with another message.

Messages may be added to a Mailbox instance using the set-like method add() and removed using a del statement or the set-like methods remove() and discard().

Mailbox interface semantics differ from dictionary semantics in some noteworthy ways. Each time a message is requested, a new representation (typically a Message instance) is generated based upon the current state of the mailbox. Similarly, when a message is added to a Mailbox instance, the provided message representation’s contents are copied. In neither case is a reference to the message representation kept by the Mailbox instance.

The default Mailbox iterator iterates over message representations, not keys as the default dictionary iterator does.
Moreover, modification of a mailbox during iteration is safe and well-defined. Messages added to the mailbox after an iterator is created will not be seen by the iterator. Messages removed from the mailbox before the iterator yields them will be silently skipped, though using a key from an iterator may result in a KeyError exception if the corresponding message is subsequently removed.

**Warning:** Be very cautious when modifying mailboxes that might be simultaneously changed by some other process. The safest mailbox format to use for such tasks is Maildir; try to avoid using single-file formats such as mbox for concurrent writing. If you’re modifying a mailbox, you must lock it by calling the `lock()` and `unlock()` methods before reading any messages in the file or making any changes by adding or deleting a message. Failing to lock the mailbox runs the risk of losing messages or corrupting the entire mailbox.

Mailbox instances have the following methods:

**add**(message)

Add *message* to the mailbox and return the key that has been assigned to it.

Parameter *message* may be a Message instance, an `email.message.Message` instance, a string, or a file-like object (which should be open in text mode). If *message* is an instance of the appropriate format-specific Message subclass (e.g., if it’s an `mboxMessage` instance and this is an `mbox` instance), its format-specific information is used. Otherwise, reasonable defaults for format-specific information are used.

**remove**(key)

__delitem__**(key)**
**discard(key)**

Delete the message corresponding to *key* from the mailbox.

If no such message exists, a **KeyError** exception is raised if the method was called as **remove()** or **__delitem__()** but no exception is raised if the method was called as **discard()**. The behavior of **discard()** may be preferred if the underlying mailbox format supports concurrent modification by other processes.

**__setitem__(key, message)**

Replace the message corresponding to *key* with *message*. Raise a **KeyError** exception if no message already corresponds to *key*.

As with **add()**, parameter *message* may be a **Message** instance, an **email.message.Message** instance, a string, or a file-like object (which should be open in text mode). If *message* is an instance of the appropriate format-specific **Message** subclass (e.g., if it's an **mboxMessage** instance and this is an **mbox** instance), its format-specific information is used. Otherwise, the format-specific information of the message that currently corresponds to *key* is left unchanged.

**iterkeys()**

**keys()**

Return an iterator over all keys if called as **iterkeys()** or return a list of keys if called as **keys()**.

**itervalues()**

**__iter__()**

**values()**

Return an iterator over representations of all messages if
called as \texttt{itervalues()} or \texttt{\_\_iter\_()} or return a list of such representations if called as \texttt{values()}. The messages are represented as instances of the appropriate format-specific \texttt{Message} subclass unless a custom message factory was specified when the \texttt{Mailbox} instance was initialized.

\begin{quote}
\textbf{Note:} The behavior of \texttt{\_\_iter\_()} is unlike that of dictionaries, which iterate over keys.
\end{quote}

\begin{itemize}
\item \texttt{iteritems()}
\item \texttt{items()}
\end{itemize}

Return an iterator over \texttt{(key, message)} pairs, where \texttt{key} is a key and \texttt{message} is a message representation, if called as \texttt{iteritems()} or return a list of such pairs if called as \texttt{items()}. The messages are represented as instances of the appropriate format-specific \texttt{Message} subclass unless a custom message factory was specified when the \texttt{Mailbox} instance was initialized.

\begin{itemize}
\item \texttt{get(key, default=None)}
\item \texttt{\_\_getitem\_\_(key)}
\end{itemize}

Return a representation of the message corresponding to \texttt{key}. If no such message exists, \texttt{default} is returned if the method was called as \texttt{get()} and a \texttt{KeyError} exception is raised if the method was called as \texttt{\_\_getitem\_\_()}. The message is represented as an instance of the appropriate format-specific \texttt{Message} subclass unless a custom message factory was specified when the \texttt{Mailbox} instance was initialized.

\begin{itemize}
\item \texttt{get\_message(key)}
\end{itemize}

Return a representation of the message corresponding to \texttt{key} as an instance of the appropriate format-specific \texttt{Message} subclass, or raise a \texttt{KeyError} exception if no such message
exists.

get_string(key)
Return a string representation of the message corresponding to key, or raise a KeyError exception if no such message exists.

get_file(key)
Return a file-like representation of the message corresponding to key, or raise a KeyError exception if no such message exists. The file-like object behaves as if open in binary mode. This file should be closed once it is no longer needed.

Note: Unlike other representations of messages, file-like representations are not necessarily independent of the Mailbox instance that created them or of the underlying mailbox. More specific documentation is provided by each subclass.

has_key(key)
__contains__(key)
Return True if key corresponds to a message, False otherwise.

__len__()
Return a count of messages in the mailbox.

clear()
Delete all messages from the mailbox.

pop(key[, default])
Return a representation of the message corresponding to key and delete the message. If no such message exists, return
default if it was supplied or else raise a KeyError exception. The message is represented as an instance of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

popitem()
Return an arbitrary (key, message) pair, where key is a key and message is a message representation, and delete the corresponding message. If the mailbox is empty, raise a KeyError exception. The message is represented as an instance of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

update(arg)
Parameter arg should be a key-to-message mapping or an iterable of (key, message) pairs. Updates the mailbox so that, for each given key and message, the message corresponding to key is set to message as if by using __setitem__(). As with __setitem__(), each key must already correspond to a message in the mailbox or else a KeyError exception will be raised, so in general it is incorrect for arg to be a Mailbox instance.

Note: Unlike with dictionaries, keyword arguments are not supported.

flush()
Write any pending changes to the filesystem. For some Mailbox subclasses, changes are always written immediately and flush() does nothing, but you should still make a habit of calling this method.
lock()

Acquire an exclusive advisory lock on the mailbox so that other processes know not to modify it. An ExternalClashError is raised if the lock is not available. The particular locking mechanisms used depend upon the mailbox format. You should *always* lock the mailbox before making any modifications to its contents.

unlock()

Release the lock on the mailbox, if any.

close()

Flush the mailbox, unlock it if necessary, and close any open files. For some Mailbox subclasses, this method does nothing.

18.4.1.1. Maildir

```python
class mailbox.Maildir(dirname, factory=rfc822.Message, create=True)
```

A subclass of Mailbox for mailboxes in Maildir format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is None, MaildirMessage is used as the default message representation. If *create* is True, the mailbox is created if it does not exist.

It is for historical reasons that *factory* defaults to rfc822.Message and that *dirname* is named as such rather than *path*. For a Maildir instance that behaves like instances of other Mailbox subclasses, set *factory* to None.

Maildir is a directory-based mailbox format invented for the qmail mail transfer agent and now widely supported by other programs.
Messages in a Maildir mailbox are stored in separate files within a common directory structure. This design allows Maildir mailboxes to be accessed and modified by multiple unrelated programs without data corruption, so file locking is unnecessary.

Maildir mailboxes contain three subdirectories, namely: tmp, new, and cur. Messages are created momentarily in the tmp subdirectory and then moved to the new subdirectory to finalize delivery. A mail user agent may subsequently move the message to the cur subdirectory and store information about the state of the message in a special “info” section appended to its file name.

Folders of the style introduced by the Courier mail transfer agent are also supported. Any subdirectory of the main mailbox is considered a folder if '.' is the first character in its name. Folder names are represented by Maildir without the leading '. '. Each folder is itself a Maildir mailbox but should not contain other folders. Instead, a logical nesting is indicated using '.' to delimit levels, e.g., “Archived.2005.07”.

**Note:** The Maildir specification requires the use of a colon (':' ) in certain message file names. However, some operating systems do not permit this character in file names. If you wish to use a Maildir-like format on such an operating system, you should specify another character to use instead. The exclamation point ('!' ) is a popular choice. For example:

```python
import mailbox
mailbox.Maildir.colon = '!' 
```

The colon attribute may also be set on a per-instance basis.

**Maildir** instances have all of the methods of **Mailbox** in addition to the following:
list_folders()
Return a list of the names of all folders.

get_folder(folder)
Return a Maildir instance representing the folder whose name is folder. A NoSuchMailboxError exception is raised if the folder does not exist.

add_folder(folder)
Create a folder whose name is folder and return a Maildir instance representing it.

remove_folder(folder)
Delete the folder whose name is folder. If the folder contains any messages, aNotEmptyError exception will be raised and the folder will not be deleted.

clean()
Delete temporary files from the mailbox that have not been accessed in the last 36 hours. The Maildir specification says that mail-reading programs should do this occasionally.

Some Mailbox methods implemented by Maildir deserve special remarks:

add(message)
__setitem__(key, message)
update(arg)

Warning: These methods generate unique file names based upon the current process ID. When using multiple threads, undetected name clashes may occur and cause corruption of the mailbox unless threads are coordinated to avoid using these methods to manipulate the same mailbox simultaneously.
flush()
   All changes to Maildir mailboxes are immediately applied, so this method does nothing.

lock()
unlock()
   Maildir mailboxes do not support (or require) locking, so these methods do nothing.

close()
   Maildir instances do not keep any open files and the underlying mailboxes do not support locking, so this method does nothing.

get_file(key)
   Depending upon the host platform, it may not be possible to modify or remove the underlying message while the returned file remains open.

See also:

maildir man page from qmail
   The original specification of the format.

Using maildir format
   Notes on Maildir by its inventor. Includes an updated name-creation scheme and details on “info” semantics.

maildir man page from Courier
   Another specification of the format. Describes a common extension for supporting folders.

18.4.1.2. mbox
class mailbox. **mbox**(path, factory=None, create=True)

A subclass of `Mailbox` for mailboxes in mbox format. Parameter `factory` is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If `factory` is `None`, `mboxMessage` is used as the default message representation. If `create` is `True`, the mailbox is created if it does not exist.

The mbox format is the classic format for storing mail on Unix systems. All messages in an mbox mailbox are stored in a single file with the beginning of each message indicated by a line whose first five characters are “From ”.

Several variations of the mbox format exist to address perceived shortcomings in the original. In the interest of compatibility, `mbox` implements the original format, which is sometimes referred to as `mboxo`. This means that the `Content-Length` header, if present, is ignored and that any occurrences of “From ” at the beginning of a line in a message body are transformed to “>From ” when storing the message, although occurrences of “>From ” are not transformed to “From ” when reading the message.

Some `Mailbox` methods implemented by `mbox` deserve special remarks:

**get_file**(key)

Using the file after calling `flush()` or `close()` on the `mbox` instance may yield unpredictable results or raise an exception.

**lock()**

**unlock()**

Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.
See also:

**mbox man page from qmail**
A specification of the format and its variations.

**mbox man page from tin**
Another specification of the format, with details on locking.

**Configuring Netscape Mail on Unix: Why The Content-Length Format is Bad**
An argument for using the original mbox format rather than a variation.

**“mbox” is a family of several mutually incompatible mailbox formats**
A history of mbox variations.

---

### 18.4.1.3. MH

```python
class mailbox. MH(path, factory=None, create=True)
```

A subclass of `Mailbox` for mailboxes in MH format. Parameter `factory` is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If `factory` is `None`, `MHMessage` is used as the default message representation. If `create` is `True`, the mailbox is created if it does not exist.

MH is a directory-based mailbox format invented for the MH Message Handling System, a mail user agent. Each message in an MH mailbox resides in its own file. An MH mailbox may contain other MH mailboxes (called `folders`) in addition to messages. Folders may be nested indefinitely. MH mailboxes also support `sequences`, which are named lists used to logically group messages without moving them to sub-folders. Sequences are defined in a file called `.mh_sequences` in each folder.
The `MH` class manipulates MH mailboxes, but it does not attempt to emulate all of `mh`'s behaviors. In particular, it does not modify and is not affected by the `context` or `.mh_profile` files that are used by `mh` to store its state and configuration.

`MH` instances have all of the methods of `Mailbox` in addition to the following:

`list_folders()`  
Return a list of the names of all folders.

`get_folder(folder)`  
Return an `MH` instance representing the folder whose name is `folder`. A `NoSuchMailboxError` exception is raised if the folder does not exist.

`add_folder(folder)`  
Create a folder whose name is `folder` and return an `MH` instance representing it.

`remove_folder(folder)`  
Delete the folder whose name is `folder`. If the folder contains any messages, a `NotEmptyError` exception will be raised and the folder will not be deleted.

`get_sequences()`  
Return a dictionary of sequence names mapped to key lists. If there are no sequences, the empty dictionary is returned.

`set_sequences(sequences)`  
Re-define the sequences that exist in the mailbox based upon `sequences`, a dictionary of names mapped to key lists, like returned by `get_sequences()`.
pack()
 Rename messages in the mailbox as necessary to eliminate gaps in numbering. Entries in the sequences list are updated correspondingly.

**Note:** Already-issued keys are invalidated by this operation and should not be subsequently used.

Some Mailbox methods implemented by MH deserve special remarks:

```python
remove(key)
__delitem__(key)
discard(key)
```
These methods immediately delete the message. The MH convention of marking a message for deletion by prepending a comma to its name is not used.

```python
lock()
unlock()
```
Three locking mechanisms are used—dot locking and, if available, the flock() and lockf() system calls. For MH mailboxes, locking the mailbox means locking the .mh_sequences file and, only for the duration of any operations that affect them, locking individual message files.

```python
get_file(key)
```
Depending upon the host platform, it may not be possible to remove the underlying message while the returned file remains open.

```python
flush()
```
All changes to MH mailboxes are immediately applied, so this method does nothing.
close()

MH instances do not keep any open files, so this method is equivalent to unlock().

See also:

nmh - Message Handling System
Home page of nmh, an updated version of the original mh.

MH & nmh: Email for Users & Programmers
A GPL-licensed book on mh and nmh, with some information on the mailbox format.

18.4.1.4. Babyl

class mailbox. Babyl(path, factory=None, create=True)
A subclass of Mailbox for mailboxes in Babyl format. Parameter factory is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If factory is None, BabylMessage is used as the default message representation. If create is True, the mailbox is created if it does not exist.

Babyl is a single-file mailbox format used by the Rmail mail user agent included with Emacs. The beginning of a message is indicated by a line containing the two characters Control-Underscore (\037) and Control-L (\014). The end of a message is indicated by the start of the next message or, in the case of the last message, a line containing a Control-Underscore (\037) character.

Messages in a Babyl mailbox have two sets of headers, original headers and so-called visible headers. Visible headers are typically a subset of the original headers that have been
reformatted or abridged to be more attractive. Each message in a Babyl mailbox also has an accompanying list of *labels*, or short strings that record extra information about the message, and a list of all user-defined labels found in the mailbox is kept in the Babyl options section.

Babyl instances have all of the methods of *Mailbox* in addition to the following:

**get_labels()**
Return a list of the names of all user-defined labels used in the mailbox.

**Note:** The actual messages are inspected to determine which labels exist in the mailbox rather than consulting the list of labels in the Babyl options section, but the Babyl section is updated whenever the mailbox is modified.

Some *Mailbox* methods implemented by Babyl deserve special remarks:

**get_file(key)**
In Babyl mailboxes, the headers of a message are not stored contiguously with the body of the message. To generate a file-like representation, the headers and body are copied together into a *StringIO* instance (from the *StringIO* module), which has an API identical to that of a file. As a result, the file-like object is truly independent of the underlying mailbox but does not save memory compared to a string representation.

**lock()**
**unlock()**
Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.
See also:

**Format of Version 5 Babyl Files**
A specification of the Babyl format.

**Reading Mail with Rmail**
The Rmail manual, with some information on Babyl semantics.

### 18.4.1.5. MMDF

class mailbox. **MMDF**(path, factory=None, create=True)

A subclass of Mailbox for mailboxes in MMDF format. Parameter `factory` is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If `factory` is `None`, **MMDF**Message is used as the default message representation. If `create` is `True`, the mailbox is created if it does not exist.

MMDF is a single-file mailbox format invented for the Multichannel Memorandum Distribution Facility, a mail transfer agent. Each message is in the same form as an mbox message but is bracketed before and after by lines containing four Control-A (`\001`) characters. As with the mbox format, the beginning of each message is indicated by a line whose first five characters are “From “, but additional occurrences of “From “ are not transformed to “>From “ when storing messages because the extra message separator lines prevent mistaking such occurrences for the starts of subsequent messages.

Some **Mailbox** methods implemented by **MMDF** deserve special remarks:

**get_file**(key)

Using the file after calling **flush()** or **close()** on the **MMDF**
instance may yield unpredictable results or raise an exception.

`lock()`
`unlock()`

Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.

See also:

**mmdf man page from tin**
A specification of MMDF format from the documentation of tin, a newsreader.

**MMDF**
A Wikipedia article describing the Multichannel Memorandum Distribution Facility.
18.4.2. Message objects

```
class mailbox.Message([message])
```


If `message` is omitted, the new instance is created in a default, empty state. If `message` is an `email.message.Message` instance, its contents are copied; furthermore, any format-specific information is converted insofar as possible if `message` is a `Message` instance. If `message` is a string or a file, it should contain an RFC 2822-compliant message, which is read and parsed.

The format-specific state and behaviors offered by subclasses vary, but in general it is only the properties that are not specific to a particular mailbox that are supported (although presumably the properties are specific to a particular mailbox format). For example, file offsets for single-file mailbox formats and file names for directory-based mailbox formats are not retained, because they are only applicable to the original mailbox. But state such as whether a message has been read by the user or marked as important is retained, because it applies to the message itself.

There is no requirement that `Message` instances be used to represent messages retrieved using `Mailbox` instances. In some situations, the time and memory required to generate `Message` representations might not be acceptable. For such situations, `Mailbox` instances also offer string and file-like representations, and a custom message factory may be specified when a `Mailbox` instance is initialized.

18.4.2.1. MaildirMessage
class mailbox.MaildirMessage([message])

A message with Maildir-specific behaviors. Parameter message has the same meaning as with the Message constructor.

Typically, a mail user agent application moves all of the messages in the new subdirectory to the cur subdirectory after the first time the user opens and closes the mailbox, recording that the messages are old whether or not they've actually been read. Each message in cur has an “info” section added to its file name to store information about its state. (Some mail readers may also add an “info” section to messages in new.) The “info” section may take one of two forms: it may contain “2,” followed by a list of standardized flags (e.g., “2,FR”) or it may contain “1,” followed by so-called experimental information. Standard flags for Maildir messages are as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Draft</td>
<td>Under composition</td>
</tr>
<tr>
<td>F</td>
<td>Flagged</td>
<td>Marked as important</td>
</tr>
<tr>
<td>P</td>
<td>Passed</td>
<td>Forwarded, resent, or bounced</td>
</tr>
<tr>
<td>R</td>
<td>Replied</td>
<td>Replied to</td>
</tr>
<tr>
<td>S</td>
<td>Seen</td>
<td>Read</td>
</tr>
<tr>
<td>T</td>
<td>Trashed</td>
<td>Marked for subsequent deletion</td>
</tr>
</tbody>
</table>

MaildirMessage instances offer the following methods:

get_subdir()

Return either “new” (if the message should be stored in the new subdirectory) or “cur” (if the message should be stored in the cur subdirectory).

**Note:** A message is typically moved from new to cur after its mailbox has been accessed, whether or not the message
is has been read. A message msg has been read if "S" in msg.get_flags() is True.

**set_subdir(subdir)**
Set the subdirectory the message should be stored in. Parameter subdir must be either “new” or “cur”.

**get_flags()**
Return a string specifying the flags that are currently set. If the message complies with the standard Maildir format, the result is the concatenation in alphabetical order of zero or one occurrence of each of 'D', 'F', 'P', 'R', 'S', and 'T'. The empty string is returned if no flags are set or if “info” contains experimental semantics.

**set_flags(flags)**
Set the flags specified by flags and unset all others.

**add_flag(flag)**
Set the flag(s) specified by flag without changing other flags. To add more than one flag at a time, flag may be a string of more than one character. The current “info” is overwritten whether or not it contains experimental information rather than flags.

**remove_flag(flag)**
Unset the flag(s) specified by flag without changing other flags. To remove more than one flag at a time, flag maybe a string of more than one character. If “info” contains experimental information rather than flags, the current “info” is not modified.

**get_date()**
Return the delivery date of the message as a floating-point
number representing seconds since the epoch.

**set_date**(date)
Set the delivery date of the message to *date*, a floating-point number representing seconds since the epoch.

**get_info**()
Return a string containing the “info” for a message. This is useful for accessing and modifying “info” that is experimental (i.e., not a list of flags).

**set_info**(info)
Set “info” to *info*, which should be a string.

When a MaildirMessage instance is created based upon an mboxMessage or MMDFMessage instance, the Status and X-Status headers are omitted and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>mboxMessage or MMDFMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“cur” subdirectory</td>
<td>O flag</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>R flag</td>
<td>A flag</td>
</tr>
<tr>
<td>S flag</td>
<td>R flag</td>
</tr>
<tr>
<td>T flag</td>
<td>D flag</td>
</tr>
</tbody>
</table>

When a MaildirMessage instance is created based upon an MHMessage instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MHMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“cur” subdirectory</td>
<td>“unseen” sequence</td>
</tr>
<tr>
<td>“cur” subdirectory and S flag</td>
<td>no “unseen” sequence</td>
</tr>
<tr>
<td>F flag</td>
<td>“flagged” sequence</td>
</tr>
<tr>
<td>R flag</td>
<td>“replied” sequence</td>
</tr>
</tbody>
</table>
When a `MaildirMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>Baby1Message state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“cur” subdirectory</td>
<td>“unseen” label</td>
</tr>
<tr>
<td>“cur” subdirectory and S flag</td>
<td>no “unseen” label</td>
</tr>
<tr>
<td>P flag</td>
<td>“forwarded” or “resent” label</td>
</tr>
<tr>
<td>R flag</td>
<td>“answered” label</td>
</tr>
<tr>
<td>T flag</td>
<td>“deleted” label</td>
</tr>
</tbody>
</table>

18.4.2.2. `mboxMessage`

```python
class mailbox.mboxMessage([message])
```

A message with mbox-specific behaviors. Parameter `message` has the same meaning as with the `Message` constructor.

Messages in an mbox mailbox are stored together in a single file. The sender’s envelope address and the time of delivery are typically stored in a line beginning with “From ” that is used to indicate the start of a message, though there is considerable variation in the exact format of this data among mbox implementations. Flags that indicate the state of the message, such as whether it has been read or marked as important, are typically stored in `Status` and `X-Status` headers.

Conventional flags for mbox messages are as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Read</td>
<td>Read</td>
</tr>
<tr>
<td>O</td>
<td>Old</td>
<td>Previously detected by MUA</td>
</tr>
<tr>
<td>D</td>
<td>Deleted</td>
<td>Marked for subsequent deletion</td>
</tr>
<tr>
<td>F</td>
<td>Flagged</td>
<td>Marked as important</td>
</tr>
<tr>
<td>A</td>
<td>Answered</td>
<td>Replied to</td>
</tr>
</tbody>
</table>
The “R” and “O” flags are stored in the Status header, and the “D”, “F”, and “A” flags are stored in the X-Status header. The flags and headers typically appear in the order mentioned.

mboxMessage instances offer the following methods:

get_from()
Return a string representing the “From ” line that marks the start of the message in an mbox mailbox. The leading “From ” and the trailing newline are excluded.

set_from(from_, time_=None)
Set the “From ” line to from_, which should be specified without a leading “From ” or trailing newline. For convenience, time_ may be specified and will be formatted appropriately and appended to from_. If time_ is specified, it should be a time.struct_time instance, a tuple suitable for passing to time.strftime(), or True (to use time.gmtime()).

get_flags()
Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of 'R', 'O', 'D', 'F', and 'A'.

set_flags(flags)
Set the flags specified by flags and unset all others. Parameter flags should be the concatenation in any order of zero or more occurrences of each of 'R', 'O', 'D', 'F', and 'A'.

add_flag(flag)
Set the flag(s) specified by flag without changing other flags. To add more than one flag at a time, flag may be a string of
more than one character.

remove_flag(flag)
Unset the flag(s) specified by flag without changing other flags. To remove more than one flag at a time, flag maybe a string of more than one character.

When an mboxMessage instance is created based upon a MaildirMessage instance, a “From ” line is generated based upon the MaildirMessage instance’s delivery date, and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MaildirMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag</td>
<td>S flag</td>
</tr>
<tr>
<td>O flag</td>
<td>“cur” subdirectory</td>
</tr>
<tr>
<td>D flag</td>
<td>T flag</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>A flag</td>
<td>R flag</td>
</tr>
</tbody>
</table>

When an mboxMessage instance is created based upon a MHMessage instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MHMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag and O flag</td>
<td>no “unseen” sequence</td>
</tr>
<tr>
<td>O flag</td>
<td>“unseen” sequence</td>
</tr>
<tr>
<td>F flag</td>
<td>“flagged” sequence</td>
</tr>
<tr>
<td>A flag</td>
<td>“replied” sequence</td>
</tr>
</tbody>
</table>

When an mboxMessage instance is created based upon a BabylMessage instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>BabylMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag and O flag</td>
<td>no “unseen” label</td>
</tr>
<tr>
<td>O flag</td>
<td>“unseen” label</td>
</tr>
</tbody>
</table>
When a `Message` instance is created based upon an `MMDFMessage` instance, the “From ” line is copied and all flags directly correspond:

<table>
<thead>
<tr>
<th>D flag</th>
<th>“deleted” label</th>
</tr>
</thead>
<tbody>
<tr>
<td>A flag</td>
<td>“answered” label</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MMDFMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag</td>
<td>R flag</td>
</tr>
<tr>
<td>O flag</td>
<td>O flag</td>
</tr>
<tr>
<td>D flag</td>
<td>D flag</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>A flag</td>
<td>A flag</td>
</tr>
</tbody>
</table>

### 18.4.2.3. MHMessage

```python
class mailbox.MMMessage([message])
```

A message with MH-specific behaviors. Parameter `message` has the same meaning as with the `Message` constructor.

MH messages do not support marks or flags in the traditional sense, but they do support sequences, which are logical groupings of arbitrary messages. Some mail reading programs (although not the standard `mh` and `nmh`) use sequences in much the same way flags are used with other formats, as follows:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>unseen</td>
<td>Not read, but previously detected by MUA</td>
</tr>
<tr>
<td>replied</td>
<td>Replied to</td>
</tr>
<tr>
<td>flagged</td>
<td>Marked as important</td>
</tr>
</tbody>
</table>

`MHMessage` instances offer the following methods:

```python
def get_sequences()
```

Return a list of the names of sequences that include this
message.

set_sequences(sequences)
Set the list of sequences that include this message.

add_sequence(sequence)
Add sequence to the list of sequences that include this message.

remove_sequence(sequence)
Remove sequence from the list of sequences that include this message.

When an MHMessage instance is created based upon a MaildirMessage instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MaildirMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” sequence</td>
<td>no S flag</td>
</tr>
<tr>
<td>“replied” sequence</td>
<td>R flag</td>
</tr>
<tr>
<td>“flagged” sequence</td>
<td>F flag</td>
</tr>
</tbody>
</table>

When an MHMessage instance is created based upon an mboxMessage or MMDFMessage instance, the Status and X-Status headers are omitted and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>mboxMessage or MMDFMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” sequence</td>
<td>no R flag</td>
</tr>
<tr>
<td>“replied” sequence</td>
<td>A flag</td>
</tr>
<tr>
<td>“flagged” sequence</td>
<td>F flag</td>
</tr>
</tbody>
</table>

When an MHMessage instance is created based upon a BabylMessage instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>BabylMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” sequence</td>
<td>“unseen” label</td>
</tr>
</tbody>
</table>
class mailbox. **BabylMessage**([message])

A message with Babyl-specific behaviors. Parameter `message` has the same meaning as with the `Message` constructor.

Certain message labels, called attributes, are defined by convention to have special meanings. The attributes are as follows:

<table>
<thead>
<tr>
<th>Label</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>unseen</td>
<td>Not read, but previously detected by MUA</td>
</tr>
<tr>
<td>deleted</td>
<td>Marked for subsequent deletion</td>
</tr>
<tr>
<td>filed</td>
<td>Copied to another file or mailbox</td>
</tr>
<tr>
<td>answered</td>
<td>Replied to</td>
</tr>
<tr>
<td>forwarded</td>
<td>Forwarded</td>
</tr>
<tr>
<td>edited</td>
<td>Modified by the user</td>
</tr>
<tr>
<td>resent</td>
<td>Resent</td>
</tr>
</tbody>
</table>

By default, Rmail displays only visible headers. The `BabylMessage` class, though, uses the original headers because they are more complete. Visible headers may be accessed explicitly if desired.

`BabylMessage` instances offer the following methods:

**get_labels()**

Return a list of labels on the message.

**set_labels(labels)**

Set the list of labels on the message to `labels`. 
add_label(label)
Add label to the list of labels on the message.

remove_label(label)
Remove label from the list of labels on the message.

get_visible()
Return an Message instance whose headers are the message’s visible headers and whose body is empty.

set_visible(visible)
Set the message’s visible headers to be the same as the headers in message. Parameter visible should be a Message instance, an email.message.Message instance, a string, or a file-like object (which should be open in text mode).

update_visible()
When a BabylMessage instance’s original headers are modified, the visible headers are not automatically modified to correspond. This method updates the visible headers as follows: each visible header with a corresponding original header is set to the value of the original header, each visible header without a corresponding original header is removed, and any of Date, From, Reply-To, To, CC, and Subject that are present in the original headers but not the visible headers are added to the visible headers.

When a BabylMessage instance is created based upon a MaildirMessage instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MaildirMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” label</td>
<td>no S flag</td>
</tr>
<tr>
<td>“deleted” label</td>
<td>T flag</td>
</tr>
<tr>
<td>“answered” label</td>
<td>R flag</td>
</tr>
</tbody>
</table>
When a `BabylMessage` instance is created based upon an `mboxMessage` or `MMDFMessage` instance, the `Status` and `X-Status` headers are omitted and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th><code>mboxMessage</code> or <code>MMDFMessage</code> state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” label</td>
<td>no R flag</td>
</tr>
<tr>
<td>“deleted” label</td>
<td>D flag</td>
</tr>
<tr>
<td>“answered” label</td>
<td>A flag</td>
</tr>
</tbody>
</table>

When a `BabylMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th><code>MHMessage</code> state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” label</td>
<td>“unseen” sequence</td>
</tr>
<tr>
<td>“answered” label</td>
<td>“replied” sequence</td>
</tr>
</tbody>
</table>

### 18.4.2.5. MMDFMessage

```python
class mailbox.MMDFMessage([message])
```

A message with MMDF-specific behaviors. Parameter `message` has the same meaning as with the `Message` constructor.

As with message in an mbox mailbox, MMDF messages are stored with the sender’s address and the delivery date in an initial line beginning with “From”. Likewise, flags that indicate the state of the message are typically stored in `Status` and `X-Status` headers.

Conventional flags for MMDF messages are identical to those of mbox message and are as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Read</td>
<td>Read</td>
</tr>
<tr>
<td>----</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>O</td>
<td>Old</td>
<td>Previously detected by MUA</td>
</tr>
<tr>
<td>D</td>
<td>Deleted</td>
<td>Marked for subsequent deletion</td>
</tr>
<tr>
<td>F</td>
<td>Flagged</td>
<td>Marked as important</td>
</tr>
<tr>
<td>A</td>
<td>Answered</td>
<td>Replied to</td>
</tr>
</tbody>
</table>

The “R” and “O” flags are stored in the Status header, and the “D”, “F”, and “A” flags are stored in the X-Status header. The flags and headers typically appear in the order mentioned.

**MMDFMessage** instances offer the following methods, which are identical to those offered by **mboxMessage**:

**get_from()**
Return a string representing the “From ” line that marks the start of the message in an mbox mailbox. The leading “From ” and the trailing newline are excluded.

**set_from**(from_, time_=None)
Set the “From ” line to from_, which should be specified without a leading “From ” or trailing newline. For convenience, time_ may be specified and will be formatted appropriately and appended to from_. If time_ is specified, it should be a time.struct_time instance, a tuple suitable for passing to time.strftime(), or True (to use time.gmtime()).

**get_flags()**
Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of 'R', 'O', 'D', 'F', and 'A'.

**set_flags**(flags)
Set the flags specified by flags and unset all others. Parameter flags should be the concatenation in any order of
zero or more occurrences of each of 'R', 'O', 'D', 'F', and 'A'.

add_flag(flag)
Set the flag(s) specified by flag without changing other flags. To add more than one flag at a time, flag may be a string of more than one character.

remove_flag(flag)
Unset the flag(s) specified by flag without changing other flags. To remove more than one flag at a time, flag maybe a string of more than one character.

When an MMDFMessage instance is created based upon a MaildirMessage instance, a “From ” line is generated based upon the MaildirMessage instance's delivery date, and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MaildirMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag</td>
<td>S flag</td>
</tr>
<tr>
<td>O flag</td>
<td>“cur” subdirectory</td>
</tr>
<tr>
<td>D flag</td>
<td>T flag</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>A flag</td>
<td>R flag</td>
</tr>
</tbody>
</table>

When an MMDFMessage instance is created based upon an MHMessage instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MHMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag and O flag</td>
<td>no “unseen” sequence</td>
</tr>
<tr>
<td>O flag</td>
<td>“unseen” sequence</td>
</tr>
<tr>
<td>F flag</td>
<td>“flagged” sequence</td>
</tr>
<tr>
<td>A flag</td>
<td>“replied” sequence</td>
</tr>
</tbody>
</table>
When an `MMDFMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>BabylMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag and O flag</td>
<td>no “unseen” label</td>
</tr>
<tr>
<td>O flag</td>
<td>“unseen” label</td>
</tr>
<tr>
<td>D flag</td>
<td>“deleted” label</td>
</tr>
<tr>
<td>A flag</td>
<td>“answered” label</td>
</tr>
</tbody>
</table>

When an `MMDFMessage` instance is created based upon an `mboxMessage` instance, the “From ” line is copied and all flags directly correspond:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>mboxMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag</td>
<td>R flag</td>
</tr>
<tr>
<td>O flag</td>
<td>O flag</td>
</tr>
<tr>
<td>D flag</td>
<td>D flag</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>A flag</td>
<td>A flag</td>
</tr>
</tbody>
</table>
The following exception classes are defined in the `mailbox` module:

- **`mailbox.Error`**
  - The based class for all other module-specific exceptions.

- **`mailbox.NoSuchMailboxError`**
  - Raised when a mailbox is expected but is not found, such as when instantiating a `Mailbox` subclass with a path that does not exist (and with the `create` parameter set to `False`), or when opening a folder that does not exist.

- **`mailbox.NotEmptyError`**
  - Raised when a mailbox is not empty but is expected to be, such as when deleting a folder that contains messages.

- **`mailbox.ExternalClashError`**
  - Raised when some mailbox-related condition beyond the control of the program causes it to be unable to proceed, such as when failing to acquire a lock that another program already holds a lock, or when a uniquely-generated file name already exists.

- **`mailbox.FormatError`**
  - Raised when the data in a file cannot be parsed, such as when an `MH` instance attempts to read a corrupted `.mh_sequences` file.
18.4.4. Deprecated classes and methods

Deprecated since version 2.6.

Older versions of the `mailbox` module do not support modification of mailboxes, such as adding or removing message, and do not provide classes to represent format-specific message properties. For backward compatibility, the older mailbox classes are still available, but the newer classes should be used in preference to them. The old classes have been removed in Python 3.

Older mailbox objects support only iteration and provide a single public method:

```python
oldmailbox.next()
```

Return the next message in the mailbox, created with the optional `factory` argument passed into the mailbox object’s constructor. By default this is an `rfc822.Message` object (see the `rfc822` module). Depending on the mailbox implementation the `fp` attribute of this object may be a true file object or a class instance simulating a file object, taking care of things like message boundaries if multiple mail messages are contained in a single file, etc. If no more messages are available, this method returns `None`.

Most of the older mailbox classes have names that differ from the current mailbox class names, except for `Maildir`. For this reason, the new `Maildir` class defines a `next()` method and its constructor differs slightly from those of the other new mailbox classes.

The older mailbox classes whose names are not the same as their newer counterparts are as follows:
Access to a classic Unix-style mailbox, where all messages are contained in a single file and separated by `From` (a.k.a. `From_`) lines. The file object `fp` points to the mailbox file. The optional `factory` parameter is a callable that should create new message objects. `factory` is called with one argument, `fp` by the `next()` method of the mailbox object. The default is the `rfc822.Message` class (see the `rfc822` module – and the note below).

**Note:** For reasons of this module’s internal implementation, you will probably want to open the `fp` object in binary mode. This is especially important on Windows.

For maximum portability, messages in a Unix-style mailbox are separated by any line that begins exactly with the string `'From '` (note the trailing space) if preceded by exactly two newlines. Because of the wide-range of variations in practice, nothing else on the `From_` line should be considered. However, the current implementation doesn’t check for the leading two newlines. This is usually fine for most applications.

The `UnixMailbox` class implements a more strict version of `From_` line checking, using a regular expression that usually correctly matched `From_` delimiters. It considers delimiter line to be separated by `From name time` lines. For maximum portability, use the `PortableUnixMailbox` class instead. This class is identical to `UnixMailbox` except that individual messages are separated by only `From` lines.

A less-strict version of `UnixMailbox`, which considers only the `From` at the beginning of the line separating messages. The
“name time” portion of the From line is ignored, to protect against some variations that are observed in practice. This works since lines in the message which begin with 'From' are quoted by mail handling software at delivery-time.

```python
class mailbox.MmdfMailbox(fp[, factory])
Access an MMDF-style mailbox, where all messages are contained in a single file and separated by lines consisting of 4 control-A characters. The file object fp points to the mailbox file. Optional factory is as with the UnixMailbox class.

class mailbox.MHMailbox(dirname[, factory])
Access an MH mailbox, a directory with each message in a separate file with a numeric name. The name of the mailbox directory is passed in dirname. factory is as with the UnixMailbox class.

class mailbox.BabylMailbox(fp[, factory])
Access a Babyl mailbox, which is similar to an MMDF mailbox. In Babyl format, each message has two sets of headers, the original headers and the visible headers. The original headers appear before a line containing only '*** EOOH ***' (End-Of-Original-Headers) and the visible headers appear after the EOOH line. Babyl-compliant mail readers will show you only the visible headers, and BabylMailbox objects will return messages containing only the visible headers. You'll have to do your own parsing of the mailbox file to get at the original headers. Mail messages start with the EOOH line and end with a line containing only '\037\014'. factory is as with the UnixMailbox class.
```

If you wish to use the older mailbox classes with the email module rather than the deprecated rfc822 module, you can do so as follows:
import email
import email.Errors
import mailbox

def msgfactory(fp):
    try:
        return email.message_from_file(fp)
    except email.Errors.MessageParseError:
        # Don't return None since that will
        # stop the mailbox iterator
        return ''

mbox = mailbox.UnixMailbox(fp, msgfactory)

Alternatively, if you know your mailbox contains only well-formed MIME messages, you can simplify this to:

import email
import mailbox

mbox = mailbox.UnixMailbox(fp, email.message_from_file)
A simple example of printing the subjects of all messages in a mailbox that seem interesting:

```python
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']  # Could possibly be None
    if subject and 'python' in subject.lower():
        print subject
```

To copy all mail from a Babyl mailbox to an MH mailbox, converting all of the format-specific information that can be converted:

```python
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

This example sorts mail from several mailing lists into different mailboxes, being careful to avoid mail corruption due to concurrent modification by other programs, mail loss due to interruption of the program, or premature termination due to malformed messages in the mailbox:

```python
import mailbox
import email.errors
list_names = ('python-list', 'python-dev', 'python-bugs')
boxes = dict((name, mailbox.mbox('~/email/%s' % name)) for name
             inbox = mailbox.Maildir('~/Maildir', factory=None)
for key in inbox.iterkeys():
    try:
        ```
message = inbox[key]
except email.errors.MessageParseError:
    continue  # The message is malformed. Just

for name in list_names:
    list_id = message['list-id']
    if list_id and name in list_id:
        # Get mailbox to use
        box = boxes[name]

        # Write copy to disk before removing original.
        # If there's a crash, you might duplicate a message
        # that's better than losing a message completely.
        box.lock()
        box.add(message)
        box.flush()
        box.unlock()

        # Remove original message
        inbox.lock()
        inbox.discard(key)
        inbox.flush()
        inbox.unlock()
        break  # Found destination, so stop looking

for box in boxes.itervalues():
    box.close()
The `mhlib` module provides a Python interface to MH folders and their contents.

The module contains three basic classes, `MH`, which represents a particular collection of folders, `Folder`, which represents a single folder, and `Message`, which represents a single message.

```python
class mhlib.MH([path[, profile]])
    MH represents a collection of MH folders.

class mhlib.Folder(mh, name)
    The Folder class represents a single folder and its messages.

class mhlib.Message(folder, number[, name])
    Message objects represent individual messages in a folder. The Message class is derived from mimetools.Message.
```
18.5.1. MH Objects

MH instances have the following methods:

MH. `error(format[, ...])`
Print an error message – can be overridden.

MH. `getprofile(key)`
Return a profile entry (`None` if not set).

MH. `getpath()`
Return the mailbox pathname.

MH. `getcontext()`
Return the current folder name.

MH. `setcontext(name)`
Set the current folder name.

MH. `listfolders()`
Return a list of top-level folders.

MH. `listallfolders()`
Return a list of all folders.

MH. `listsusbfolders(name)`
Return a list of direct subfolders of the given folder.

MH. `listallsusbfolders(name)`
Return a list of all subfolders of the given folder.

MH. `makefolder(name)`
Create a new folder.
deletefolder(name)
Delete a folder – must have no subfolders.

openfolder(name)
Return a new open folder object.
18.5.2. Folder Objects

Folder instances represent open folders and have the following methods:

Folder. `error(format[, ...])`
Print an error message – can be overridden.

Folder. `getfullname()`
Return the folder’s full pathname.

Folder. `getsequencesfilename()`
Return the full pathname of the folder’s sequences file.

Folder. `getmessagefilename(n)`
Return the full pathname of message n of the folder.

Folder. `listmessages()`
Return a list of messages in the folder (as numbers).

Folder. `getcurrent()`
Return the current message number.

Folder. `setcurrent(n)`
Set the current message number to n.

Folder. `parsesequence(seq)`
Parse msgs syntax into list of messages.

Folder. `getlast()`
Get last message, or ø if no messages are in the folder.

Folder. `setlast(n)`
Set last message (internal use only).
Folder. **getsequences()**

Return dictionary of sequences in folder. The sequence names are used as keys, and the values are the lists of message numbers in the sequences.

Folder. **putsequences(dict)**

Return dictionary of sequences in folder name: list.

Folder. **removemessages(list)**

Remove messages in list from folder.

Folder. **refilemessages(list, tofolder)**

Move messages in list to other folder.

Folder. **movemessage(n, tofolder, ton)**

Move one message to a given destination in another folder.

Folder. **copymessage(n, tofolder, ton)**

Copy one message to a given destination in another folder.
18.5.3. Message Objects

The `Message` class adds one method to those of `mimetools.Message`:

Message. `openmessage(n)`

Return a new open message object (costs a file descriptor).
18.6. mimetools — Tools for parsing MIME messages

 Deprecated since version 2.3: The email package should be used in preference to the mimetools module. This module is present only to maintain backward compatibility, and it has been removed in 3.x.

This module defines a subclass of the rfc822 module's Message class and a number of utility functions that are useful for the manipulation for MIME multipart or encoded message.

It defines the following items:

 class mimetools.Message(fp[, seekable])

 Return a new instance of the Message class. This is a subclass of the rfc822.Message class, with some additional methods (see below). The seekable argument has the same meaning as for rfc822.Message.

 mimetools.choose_boundary()

 Return a unique string that has a high likelihood of being usable as a part boundary. The string has the form 'hostipaddr.uid.pid.timestamp.random'.

 mimetools.decode(input, output, encoding)

 Read data encoded using the allowed MIME encoding from open file object input and write the decoded data to open file object output. Valid values for encoding include 'base64', 'quoted-printable', 'uuencode', 'x-uuencode', 'uue', 'x-uue', '7bit', and '8bit'. Decoding messages encoded in '7bit' or '8bit' has no effect. The input is simply copied to the output.
**mimetools.** encode\((input, output, encoding)\)

Read data from open file object \(input\) and write it encoded using the allowed MIME \(encoding\) to open file object \(output\). Valid values for \(encoding\) are the same as for \(decode()\).

**mimetools.** copyliteral\((input, output)\)

Read lines from open file \(input\) until EOF and write them to open file \(output\).

**mimetools.** copybinary\((input, output)\)

Read blocks until EOF from open file \(input\) and write them to open file \(output\). The block size is currently fixed at 8192.

**See also:**

**Module** email

Comprehensive email handling package; supersedes the mimetools module.

**Module** rfc822

Provides the base class for mimetools.Message.

**Module** multifile

Support for reading files which contain distinct parts, such as MIME data.

http://faqs.cs.uu.nl/na-dir/mail/mime-faq/.html

The MIME Frequently Asked Questions document. For an overview of MIME, see the answer to question 1.1 in Part 1 of this document.
18.6.1. Additional Methods of Message Objects

The `Message` class defines the following methods in addition to the `rfc822.Message` methods:

**Message.getplist()**

Return the parameter list of the `Content-Type` header. This is a list of strings. For parameters of the form `key=value`, `key` is converted to lower case but `value` is not. For example, if the message contains the header `Content-type: text/html; spam=1; Spam=2; Spam` then `getplist()` will return the Python list `['spam=1', 'spam=2', 'Spam']`.

**Message.getparam(name)**

Return the `value` of the first parameter (as returned by `getplist()`) of the form `name=value` for the given `name`. If `value` is surrounded by quotes of the form `'<...>'` or `"..."`, these are removed.

**Message.getencoding()**

Return the encoding specified in the `Content-Transfer-Encoding` message header. If no such header exists, return `'7bit'`. The encoding is converted to lower case.

**Message.gettype()**

Return the message type (of the form `type/subtype`) as specified in the `Content-Type` header. If no such header exists, return `'text/plain'`. The type is converted to lower case.

**Message.getmaintype()**

Return the main type as specified in the `Content-Type` header.
no such header exists, return 'text'. The main type is converted to lower case.

Message. \texttt{getsubtype()}

Return the subtype as specified in the \texttt{Content-Type} header. If no such header exists, return 'plain'. The subtype is converted to lower case.
18.7. mimetypes — Map filenames to MIME types

Source code: Lib/mimetypes.py

The mimetypes module converts between a filename or URL and the MIME type associated with the filename extension. Conversions are provided from filename to MIME type and from MIME type to filename extension; encodings are not supported for the latter conversion.

The module provides one class and a number of convenience functions. The functions are the normal interface to this module, but some applications may be interested in the class as well.

The functions described below provide the primary interface for this module. If the module has not been initialized, they will call init() if they rely on the information init() sets up.

mimetypes. guess_type(url, strict=True)

Guess the type of a file based on its filename or URL, given by url. The return value is a tuple (type, encoding) where type is None if the type can’t be guessed (missing or unknown suffix) or a string of the form 'type/subtype', usable for a MIME content-type header.

encoding is None for no encoding or the name of the program used to encode (e.g. compress or gzip). The encoding is suitable for use as a Content-Encoding header, not as a Content-Transfer-Encoding header. The mappings are table driven. Encoding suffixes are case sensitive; type suffixes are first tried case sensitively, then case insensitively.
The optional *strict* argument is a flag specifying whether the list of known MIME types is limited to only the official types registered with IANA. When *strict* is `True` (the default), only the IANA types are supported; when *strict* is `False`, some additional non-standard but commonly used MIME types are also recognized.

```python
mimetypes.guess_all_extensions(type, strict=True)
```

Guess the extensions for a file based on its MIME type, given by *type*. The return value is a list of strings giving all possible filename extensions, including the leading dot (`'.'`). The extensions are not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by `guess_type()`.

The optional *strict* argument has the same meaning as with the `guess_type()` function.

```python
mimetypes.guess_extension(type, strict=True)
```

Guess the extension for a file based on its MIME type, given by *type*. The return value is a string giving a filename extension, including the leading dot (`'.'`). The extension is not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by `guess_type()`. If no extension can be guessed for *type*, `None` is returned.

The optional *strict* argument has the same meaning as with the `guess_type()` function.

Some additional functions and data items are available for controlling the behavior of the module.

```python
mimetypes.init(files=None)
```

Initialize the internal data structures. If given, *files* must be a sequence of file names which should be used to augment the
default type map. If omitted, the file names to use are taken from knownfiles; on Windows, the current registry settings are loaded. Each file named in files or knownfiles takes precedence over those named before it. Calling init() repeatedly is allowed.

Specifying an empty list for files will prevent the system defaults from being applied: only the well-known values will be present from a built-in list.

*Changed in version 2.7:* Previously, Windows registry settings were ignored.

mimetypes.read_mime_types(filename)
Load the type map given in the file filename, if it exists. The type map is returned as a dictionary mapping filename extensions, including the leading dot ('.'), to strings of the form 'type/subtype'. If the file filename does not exist or cannot be read, None is returned.

mimetypes.add_type(type, ext, strict=True)
Add a mapping from the MIME type type to the extension ext. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

When strict is True (the default), the mapping will be added to the official MIME types, otherwise to the non-standard ones.

mimetypes.init
Flag indicating whether or not the global data structures have been initialized. This is set to True by init().

mimetypes.knownfiles
List of type map file names commonly installed. These files are typically named mime.types and are installed in different locations
by different packages.

mimetypes.**suffix_map**

Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately.

mimetypes.**encodings_map**

Dictionary mapping filename extensions to encoding types.

mimetypes.**types_map**

Dictionary mapping filename extensions to MIME types.

mimetypes.**common_types**

Dictionary mapping filename extensions to non-standard, but commonly found MIME types.

An example usage of the module:

```python
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ...
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```
18.7.1. MimeTypes Objects

The `MimeTypes` class may be useful for applications which may want more than one MIME-type database; it provides an interface similar to the one of the `mimetypes` module.

```python
class mimetypes.MimeTypes(filenames=(), strict=True)
```

This class represents a MIME-types database. By default, it provides access to the same database as the rest of this module. The initial database is a copy of that provided by the module, and may be extended by loading additional `mime.types`-style files into the database using the `read()` or `readfp()` methods. The mapping dictionaries may also be cleared before loading additional data if the default data is not desired.

The optional `filenames` parameter can be used to cause additional files to be loaded “on top” of the default database.

`MimeTypes.suffix_map`

Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately. This is initially a copy of the global `suffix_map` defined in the module.

`MimeTypes.encodings_map`

Dictionary mapping filename extensions to encoding types. This is initially a copy of the global `encodings_map` defined in the module.

`MimeTypes.types_map`

Tuple containing two dictionaries, mapping filename extensions
to MIME types: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by common_types and types_map.

MimeTypes.types_map_inv
Tuple containing two dictionaries, mapping MIME types to a list of filename extensions: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by common_types and types_map.

MimeTypes.guess_extension(type, strict=True)
Similar to the guess_extension() function, using the tables stored as part of the object.

MimeTypes.guess_type(url, strict=True)
Similar to the guess_type() function, using the tables stored as part of the object.

MimeTypes.guess_all_extensions(type, strict=True)
Similar to the guess_all_extensions() function, using the tables stored as part of the object.

MimeTypes.read(filename, strict=True)
Load MIME information from a file named filename. This uses readfp() to parse the file.

If strict is True, information will be added to list of standard types, else to the list of non-standard types.

MimeTypes.readfp(fp, strict=True)
Load MIME type information from an open file fp. The file must have the format of the standard mime.types files.

If strict is True, information will be added to the list of standard types, else to the list of non-standard types.
MimeTypes. **read_windows_registry**(\texttt{strict=True})

Load MIME type information from the Windows registry. Availability: Windows.

If \texttt{strict} is \texttt{True}, information will be added to the list of standard types, else to the list of non-standard types.

*New in version 2.7.*
18.8. MimeWriter — Generic MIME file writer

_Deprecated since version 2.3:_ The `email` package should be used in preference to the `MimeWriter` module. This module is present only to maintain backward compatibility.

This module defines the class `MimeWriter`. The `MimeWriter` class implements a basic formatter for creating MIME multi-part files. It doesn’t seek around the output file nor does it use large amounts of buffer space. You must write the parts out in the order that they should occur in the final file. `MimeWriter` does buffer the headers you add, allowing you to rearrange their order.

```python
class MimeWriter:
    def __init__(self, fp):
        # Return a new instance of the MimeWriter class. The only argument passed, fp, is a file object to be used for writing. Note that a StringIO object could also be used.
```


18.8.1. MimeWriter Objects

MimeWriter instances have the following methods:

MimeWriter.addheader(key, value[, prefix])
Add a header line to the MIME message. The key is the name of
the header, where the value obviously provides the value of the
header. The optional argument prefix determines where the
header is inserted; 0 means append at the end, 1 is insert at the
start. The default is to append.

MimeWriter.flushheaders()
Causes all headers accumulated so far to be written out (and
forgotten). This is useful if you don’t need a body part at all, e.g.
for a subpart of type message/rfc822 that's (mis)used to store
some header-like information.

MimeWriter.startbody(ctype[, plist[, prefix]])
Returns a file-like object which can be used to write to the body
of the message. The content-type is set to the provided ctype,
and the optional parameter plist provides additional parameters
for the content-type declaration. prefix functions as in
addheader() except that the default is to insert at the start.

MimeWriter.startmultipartbody(subtype[, boundary[, plist[, prefix]]])
Returns a file-like object which can be used to write to the body
of the message. Additionally, this method initializes the multi-part
code, where subtype provides the multipart subtype, boundary
may provide a user-defined boundary specification, and plist
provides optional parameters for the subtype. prefix functions as
in startbody(). Subparts should be created using nextpart().
**MimeWriter.nextpart()**

Returns a new instance of `MimeWriter` which represents an individual part in a multipart message. This may be used to write the part as well as used for creating recursively complex multipart messages. The message must first be initialized with `startmultipartbody()` before using `nextpart()`.

**MimeWriter.lastpart()**

This is used to designate the last part of a multipart message, and should *always* be used when writing multipart messages.
### 18.9. mimify — MIME processing of mail messages

**Deprecated since version 2.3:** The `email` package should be used in preference to the `mimify` module. This module is present only to maintain backward compatibility.

The `mimify` module defines two functions to convert mail messages to and from MIME format. The mail message can be either a simple message or a so-called multipart message. Each part is treated separately. Mimifying (a part of) a message entails encoding the message as quoted-printable if it contains any characters that cannot be represented using 7-bit ASCII. Unmimifying (a part of) a message entails undoing the quoted-printable encoding. Mimify and unmimify are especially useful when a message has to be edited before being sent. Typical use would be:

```
unmimify message
edit message
mimify message
send message
```

The modules defines the following user-callable functions and user-settable variables:

**mimify.**

`mimify(infile, outfile)`

Copy the message in `infile` to `outfile`, converting parts to quoted-printable and adding MIME mail headers when necessary. `infile` and `outfile` can be file objects (actually, any object that has a `readline()` method (for `infile`) or a `write()` method (for `outfile`)) or strings naming the files. If `infile` and `outfile` are both strings, they may have the same value.
mimify.\texttt{unmimify(infile, outfile[, decode\_base64])}

Copy the message in \texttt{infile} to \texttt{outfile}, decoding all quoted-printable parts. \texttt{infile} and \texttt{outfile} can be file objects (actually, any object that has a \texttt{readline()} method (for \texttt{infile}) or a \texttt{write()} method (for \texttt{outfile})) or strings naming the files. If \texttt{infile} and \texttt{outfile} are both strings, they may have the same value. If the \texttt{decode\_base64} argument is provided and tests true, any parts that are coded in the base64 encoding are decoded as well.

mimify.\texttt{mime\_decode\_header(line)}

Return a decoded version of the encoded header line in \texttt{line}. This only supports the ISO 8859-1 charset (Latin-1).

mimify.\texttt{mime\_encode\_header(line)}

Return a MIME-encoded version of the header line in \texttt{line}.

mimify.\texttt{MAXLEN}

By default, a part will be encoded as quoted-printable when it contains any non-ASCII characters (characters with the 8th bit set), or if there are any lines longer than \texttt{MAXLEN} characters (default value 200).

mimify.\texttt{CHARSET}

When not specified in the mail headers, a character set must be filled in. The string used is stored in \texttt{CHARSET}, and the default value is ISO-8859-1 (also known as Latin1 (latin-one)).

This module can also be used from the command line. Usage is as follows:

```plaintext
mimify.py -e [-l length] [infile [outfile]]
mimify.py -d [-b] [infile [outfile]]
```

to encode (mimify) and decode (unmimify) respectively. \texttt{infile} defaults to standard input, \texttt{outfile} defaults to standard output. The
same file can be specified for input and output.

If the -I option is given when encoding, if there are any lines longer than the specified length, the containing part will be encoded.

If the -b option is given when decoding, any base64 parts will be decoded as well.

See also:

Module quopri
Encode and decode MIME quoted-printable files.
18.10. multifile — Support for files containing distinct parts

*Deprecated since version 2.5:* The `email` package should be used in preference to the `multifile` module. This module is present only to maintain backward compatibility.

The `MultiFile` object enables you to treat sections of a text file as file-like input objects, with `''` being returned by `readline()` when a given delimiter pattern is encountered. The defaults of this class are designed to make it useful for parsing MIME multipart messages, but by subclassing it and overriding methods it can be easily adapted for more general use.

```python
class multifile.MultiFile(fp[, seekable])

Create a multi-file. You must instantiate this class with an input object argument for the `MultiFile` instance to get lines from, such as a file object returned by `open()`.

`MultiFile` only ever looks at the input object’s `readline()`, `seek()` and `tell()` methods, and the latter two are only needed if you want random access to the individual MIME parts. To use `MultiFile` on a non-seekable stream object, set the optional `seekable` argument to false; this will prevent using the input object’s `seek()` and `tell()` methods.

It will be useful to know that in `MultiFile`'s view of the world, text is composed of three kinds of lines: data, section-dividers, and end-markers. `MultiFile` is designed to support parsing of messages that may have multiple nested message parts, each with its own pattern for section-divider and end-marker lines.
See also:

**Module email**

Comprehensive email handling package; supersedes the `multifile` module.
18.10.1. MultiFile Objects

A MultiFile instance has the following methods:

MultiFile. **readline**(str)
Read a line. If the line is data (not a section-divider or end-marker or real EOF) return it. If the line matches the most-recently-stacked boundary, return '' and set self.last to 1 or 0 according as the match is or is not an end-marker. If the line matches any other stacked boundary, raise an error. On encountering end-of-file on the underlying stream object, the method raises Error unless all boundaries have been popped.

MultiFile. **readlines**(str)
Return all lines remaining in this part as a list of strings.

MultiFile. **read**()
Read all lines, up to the next section. Return them as a single (multiline) string. Note that this doesn’t take a size argument!

MultiFile. **seek**(pos[, whence])
Seek. Seek indices are relative to the start of the current section. The pos and whence arguments are interpreted as for a file seek.

MultiFile. **tell**()
Return the file position relative to the start of the current section.

MultiFile. **next**()
Skip lines to the next section (that is, read lines until a section-divider or end-marker has been consumed). Return true if there is such a section, false if an end-marker is seen. Re-enable the most-recently-pushed boundary.

MultiFile. **is_data**(str)
Return true if \textit{str} is data and false if it might be a section boundary. As written, it tests for a prefix other than '\-\-' at start of line (which all MIME boundaries have) but it is declared so it can be overridden in derived classes.

Note that this test is used intended as a fast guard for the real boundary tests; if it always returns false it will merely slow processing, not cause it to fail.

\textbf{MultiFile}.\texttt{push(str)}

Push a boundary string. When a decorated version of this boundary is found as an input line, it will be interpreted as a section-divider or end-marker (depending on the decoration, see \textit{RFC 2045}). All subsequent reads will return the empty string to indicate end-of-file, until a call to \texttt{pop()} removes the boundary a or next() call reenables it.

It is possible to push more than one boundary. Encountering the most-recently-pushed boundary will return EOF; encountering any other boundary will raise an error.

\textbf{MultiFile}.\texttt{pop()}

Pop a section boundary. This boundary will no longer be interpreted as EOF.

\textbf{MultiFile}.\texttt{section_divider(str)}

Turn a boundary into a section-divider line. By default, this method prepends '\-\-' (which MIME section boundaries have) but it is declared so it can be overridden in derived classes. This method need not append LF or CR-LF, as comparison with the result ignores trailing whitespace.

\textbf{MultiFile}.\texttt{end_marker(str)}

Turn a boundary string into an end-marker line. By default, this method prepends '\-\-' and appends '\-\-' (like a MIME-multipart
end-of-message marker) but it is declared so it can be overridden in derived classes. This method need not append LF or CR-LF, as comparison with the result ignores trailing whitespace.

Finally, `MultiFile` instances have two public instance variables:

`MultiFile.level`
Nesting depth of the current part.

`MultiFile.last`
True if the last end-of-file was for an end-of-message marker.
18.10.2. MultiFile Example

```python
import mimetools
import multifile
import StringIO

def extract_mime_part_matching(stream, mimetype):
    """Return the first element in a multipart MIME message on stream
    matching mimetype.""

    msg = mimetools.Message(stream)
    msgtype = msg.gettype()
    params = msg.getplist()

    data = StringIO.StringIO()
    if msgtype[:10] == "multipart/":

        file = multifile.MultiFile(stream)
        file.push(msg.getparam("boundary"))
        while file.next():
            submsg = mimetools.Message(file)
            try:
                data = StringIO.StringIO()
                mimetools.decode(file, data, submsg.getencoding)
            except ValueError:
                continue
            if submsg.gettype() == mimetype:
                break
        file.pop()
    return data.getvalue()
```

18.11. rfc822 — Parse RFC 2822 mail headers

*Deprecated since version 2.3:* The `email` package should be used in preference to the `rfc822` module. This module is present only to maintain backward compatibility, and has been removed in Python 3.

This module defines a class, `Message`, which represents an “email message” as defined by the Internet standard **RFC 2822**. [1] Such messages consist of a collection of message headers, and a message body. This module also defines a helper class `AddressList` for parsing **RFC 2822** addresses. Please refer to the RFC for information on the specific syntax of **RFC 2822** messages.

The `mailbox` module provides classes to read mailboxes produced by various end-user mail programs.

```python
class rfc822.Message(file[, seekable])
```

A `Message` instance is instantiated with an input object as parameter. `Message` relies only on the input object having a `readline()` method; in particular, ordinary file objects qualify. Instantiation reads headers from the input object up to a delimiter line (normally a blank line) and stores them in the instance. The message body, following the headers, is not consumed.

This class can work with any input object that supports a `readline()` method. If the input object has seek and tell capability, the `rewindbody()` method will work; also, illegal lines will be pushed back onto the input stream. If the input object lacks seek but has an `unread()` method that can push back a line
of input, `Message` will use that to push back illegal lines. Thus this class can be used to parse messages coming from a buffered stream.

The optional `seekable` argument is provided as a workaround for certain stdio libraries in which `tell()` discards buffered data before discovering that the `lseek()` system call doesn’t work. For maximum portability, you should set the seekable argument to zero to prevent that initial `tell()` when passing in an unseekable object such as a file object created from a socket object.

Input lines as read from the file may either be terminated by CR-LF or by a single linefeed; a terminating CR-LF is replaced by a single linefeed before the line is stored.

All header matching is done independent of upper or lower case; e.g. `m['From']`, `m['from']` and `m['FROM']` all yield the same result.

```python
class rfc822.AddressList(field)
    You may instantiate the AddressList helper class using a single string parameter, a comma-separated list of RFC 2822 addresses to be parsed. (The parameter `None` yields an empty list.)

rfc822.quote(str)
    Return a new string with backslashes in `str` replaced by two backslashes and double quotes replaced by backslash-double quote.

rfc822.unquote(str)
    Return a new string which is an unquoted version of `str`. If `str` ends and begins with double quotes, they are stripped off. Likewise if `str` ends and begins with angle brackets, they are stripped off.

rfc822.parseaddr(address)
```
Parse *address*, which should be the value of some address-containing field such as *To* or *Cc*, into its constituent “realname” and “email address” parts. Returns a tuple of that information, unless the parse fails, in which case a 2-tuple (None, None) is returned.

**rfc822.dump_address_pair(pair)**

The inverse of *parseaddr()*; this takes a 2-tuple of the form (realname, email_address) and returns the string value suitable for a *To* or *Cc* header. If the first element of *pair* is false, then the second element is returned unmodified.

**rfc822.parsedate(date)**

Attempts to parse a date according to the rules in RFC 2822. However, some mailers don’t follow that format as specified, so *parsedate()* tries to guess correctly in such cases. *date* is a string containing an RFC 2822 date, such as 'Mon, 20 Nov 1995 19:12:08 -0500'. If it succeeds in parsing the date, *parsedate()* returns a 9-tuple that can be passed directly to *time.mktime()*; otherwise None will be returned. Note that indexes 6, 7, and 8 of the result tuple are not usable.

**rfc822.parsedate_tz(date)**

Performs the same function as *parsedate()*; but returns either None or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to *time.mktime()*; and the tenth is the offset of the date’s timezone from UTC (which is the official term for Greenwich Mean Time). (Note that the sign of the timezone offset is the opposite of the sign of the *time.timezone* variable for the same timezone; the latter variable follows the POSIX standard while this module follows RFC 2822.) If the input string has no timezone, the last element of the tuple returned is None. Note that indexes 6, 7, and 8 of the result tuple are not usable.
rfc822.mktime_tz(tuple)

Turn a 10-tuple as returned by parsedate_tz() into a UTC timestamp. If the timezone item in the tuple is None, assume local time. Minor deficiency: this first interprets the first 8 elements as a local time and then compensates for the timezone difference; this may yield a slight error around daylight savings time switch dates. Not enough to worry about for common use.

See also:

Module email
Comprehensive email handling package; supersedes the rfc822 module.

Module mailbox
Classes to read various mailbox formats produced by end-user mail programs.

Module mimetools
Subclass of rfc822.Message that handles MIME encoded messages.
18.11.1. Message Objects

A `Message` instance has the following methods:

`Message.rewindbody()`
Seek to the start of the message body. This only works if the file object is seekable.

`Message.isheader(line)`
Returns a line's canonicalized fieldname (the dictionary key that will be used to index it) if the line is a legal RFC 2822 header; otherwise returns `None` (implying that parsing should stop here and the line be pushed back on the input stream). It is sometimes useful to override this method in a subclass.

`Message.islast(line)`
Return true if the given line is a delimiter on which Message should stop. The delimiter line is consumed, and the file object's read location positioned immediately after it. By default this method just checks that the line is blank, but you can override it in a subclass.

`Message.iscomment(line)`
Return `True` if the given line should be ignored entirely, just skipped. By default this is a stub that always returns `False`, but you can override it in a subclass.

`Message.getallmatchingheaders(name)`
Return a list of lines consisting of all headers matching `name`, if any. Each physical line, whether it is a continuation line or not, is a separate list item. Return the empty list if no header matches `name`.

`Message.getfirstmatchingheader(name)`
Return a list of lines comprising the first header matching \textit{name}, and its continuation line(s), if any. Return \texttt{None} if there is no header matching \textit{name}.

\textbf{Message. getrawheader(}\textit{name})

Return a single string consisting of the text after the colon in the first header matching \textit{name}. This includes leading whitespace, the trailing linefeed, and internal linefeeds and whitespace if there any continuation line(s) were present. Return \texttt{None} if there is no header matching \textit{name}.

\textbf{Message. getheader(}\textit{name}, \textit{default})

Return a single string consisting of the last header matching \textit{name}, but strip leading and trailing whitespace. Internal whitespace is not stripped. The optional \textit{default} argument can be used to specify a different default to be returned when there is no header matching \textit{name}; it defaults to \texttt{None}. This is the preferred way to get parsed headers.

\textbf{Message. get(}\textit{name}, \textit{default})

An alias for \texttt{getheader()}, to make the interface more compatible with regular dictionaries.

\textbf{Message. getaddr(}\textit{name})

Return a pair (full name, email address) parsed from the string returned by \texttt{getheader(name)}. If no header matching \textit{name} exists, return \texttt{(None, None)}; otherwise both the full name and the address are (possibly empty) strings.

Example: If \texttt{m}'s first \texttt{From} header contains the string

\texttt{'jack@cwi.nl (Jack Jansen)'}, then \texttt{m.getaddr('From')} will yield the pair \texttt{('Jack Jansen', 'jack@cwi.nl')}. If the header contained

\texttt{'Jack Jansen <jack@cwi.nl>'} instead, it would yield the exact
same result.

Message. **getaddrlist**(*name*)
This is similar to **getaddr(list)**, but parses a header containing a list of email addresses (e.g. a *To* header) and returns a list of *(full name, email address)* pairs (even if there was only one address in the header). If there is no header matching *name*, return an empty list.

If multiple headers exist that match the named header (e.g. if there are several *Cc* headers), all are parsed for addresses. Any continuation lines the named headers contain are also parsed.

Message. **getdate**(*name*)
Retrieve a header using **getheader()** and parse it into a 9-tuple compatible with **time.mktime()**; note that fields 6, 7, and 8 are not usable. If there is no header matching *name*, or it is unparsable, return None.

Date parsing appears to be a black art, and not all mailers adhere to the standard. While it has been tested and found correct on a large collection of email from many sources, it is still possible that this function may occasionally yield an incorrect result.

Message. **getdate_tz**(*name*)
Retrieve a header using **getheader()** and parse it into a 10-tuple; the first 9 elements will make a tuple compatible with **time.mktime()**, and the 10th is a number giving the offset of the date’s timezone from UTC. Note that fields 6, 7, and 8 are not usable. Similarly to **getdate()**, if there is no header matching *name*, or it is unparsable, return None.

Message instances also support a limited mapping interface. In particular: **m[name]** is like **m.getheader(name)** but raises **KeyError**
there is no matching header; and \texttt{len(m)}, \texttt{m.get(name[, default])}, \texttt{name \ in \ m}, \texttt{m.keys()}, \texttt{m.values()} \texttt{m.items()}, and \texttt{m.setdefault(name[, default])} act as expected, with the one difference that \texttt{setdefault()} uses an empty string as the default value. \texttt{Message} instances also support the mapping writable interface \texttt{m[name] = value and del m[name]}. \texttt{Message} objects do not support the \texttt{clear()}, \texttt{copy()}, \texttt{popitem()}, or \texttt{update()} methods of the mapping interface. (Support for \texttt{get()} and \texttt{setdefault()} was only added in Python 2.2.)

Finally, \texttt{Message} instances have some public instance variables:

\texttt{Message.headers}

A list containing the entire set of header lines, in the order in which they were read (except that setitem calls may disturb this order). Each line contains a trailing newline. The blank line terminating the headers is not contained in the list.

\texttt{Message.fp}

The file or file-like object passed at instantiation time. This can be used to read the message content.

\texttt{Message.unixfrom}

The Unix \texttt{From} line, if the message had one, or an empty string. This is needed to regenerate the message in some contexts, such as an \texttt{mbox}-style mailbox file.
18.11.2. AddressList Objects

An **AddressList** instance has the following methods:

**AddressList.**

- **.\_len\_\_**()
  
  Return the number of addresses in the address list.

- **.\_str\_\_**()
  
  Return a canonicalized string representation of the address list. Addresses are rendered in “name” \(<\text{host}@\text{domain}>\) form, comma-separated.

- **.\_add\_\_(alist)**
  
  Return a new **AddressList** instance that contains all addresses in both **AddressList** operands, with duplicates removed (set union).

- **.\_iadd\_\_(alist)**
  
  In-place version of **.\_add\_\_**(); turns this **AddressList** instance into the union of itself and the right-hand instance, **alist**.

- **.\_sub\_\_(alist)**
  
  Return a new **AddressList** instance that contains every address in the left-hand **AddressList** operand that is not present in the right-hand address operand (set difference).

- **.\_isub\_\_(alist)**
  
  In-place version of **.\_sub\_\_**(), removing addresses in this list which are also in **alist**.

Finally, **AddressList** instances have one public instance variable:

**AddressList.**

- **addresslist**
  
  A list of tuple string pairs, one per address. In each member, the first is the canonicalized name part, the second is the actual
route-address (@-separated username-host.domain pair).

Footnotes

This module originally conformed to RFC 822, hence the name. Since then, RFC 2822 has been released as an update to RFC 822. This module should be considered RFC 2822-conformant, especially in cases where the syntax or semantics have changed since RFC 822.

[1] RFC 822
This module provides data encoding and decoding as specified in [RFC 3548](https://tools.ietf.org/html/rfc3548). This standard defines the Base16, Base32, and Base64 algorithms for encoding and decoding arbitrary binary strings into text strings that can be safely sent by email, used as parts of URLs, or included as part of an HTTP POST request. The encoding algorithm is not the same as the [uuencode](https://tools.ietf.org/html/rfc1482) program.

There are two interfaces provided by this module. The modern interface supports encoding and decoding string objects using both base-64 alphabets defined in [RFC 3548](https://tools.ietf.org/html/rfc3548) (normal, and URL- and filesystem-safe). The legacy interface provides for encoding and decoding to and from file-like objects as well as strings, but only using the Base64 standard alphabet.

The modern interface, which was introduced in Python 2.4, provides:

```python
base64.b64encode(s[, altchars])
```

Encode a string using Base64.

`s` is the string to encode. Optional `altchars` must be a string of at least length 2 (additional characters are ignored) which specifies an alternative alphabet for the `+` and `/` characters. This allows an application to e.g. generate URL or filesystem safe Base64 strings. The default is `None`, for which the standard Base64 alphabet is used.

The encoded string is returned.
base64. **b64decode**(s[, altchars])
Decode a Base64 encoded string.

s is the string to decode. Optional altchars must be a string of at least length 2 (additional characters are ignored) which specifies the alternative alphabet used instead of the + and / characters.

The decoded string is returned. A **TypeError** is raised if s is incorrectly padded. Characters that are neither in the normal base-64 alphabet nor the alternative alphabet are discarded prior to the padding check.

base64. **standard_b64encode**(s)
Encode string s using the standard Base64 alphabet.

base64. **standard_b64decode**(s)
Decode string s using the standard Base64 alphabet.

base64. **urlsafe_b64encode**(s)
Encode string s using the URL- and filesystem-safe alphabet, which substitutes - instead of + and _ instead of / in the standard Base64 alphabet. The result can still contain =.

base64. **urlsafe_b64decode**(s)
Decode string s using the URL- and filesystem-safe alphabet, which substitutes - instead of + and _ instead of / in the standard Base64 alphabet.

base64. **b32encode**(s)
Encode a string using Base32. s is the string to encode. The encoded string is returned.

base64. **b32decode**(s[, casefold[, map01]])
Decode a Base32 encoded string.
s is the string to decode. Optional *casefold* is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is *False*.

**RFC 3548** allows for optional mapping of the digit 0 (zero) to the letter O (oh), and for optional mapping of the digit 1 (one) to either the letter I (eye) or letter L (el). The optional argument *map01* when not *None*, specifies which letter the digit 1 should be mapped to (when *map01* is not *None*, the digit 0 is always mapped to the letter O). For security purposes the default is *None*, so that 0 and 1 are not allowed in the input.

The decoded string is returned. A *TypeError* is raised if *s* is incorrectly padded or if there are non-alphabet characters present in the string.

```
base64. **b16encode**(s)
```

Encode a string using Base16.

*s* is the string to encode. The encoded string is returned.

```
base64. **b16decode**(s[, *casefold]*)
```

Decode a Base16 encoded string.

*s* is the string to decode. Optional *casefold* is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is *False*.

The decoded string is returned. A *TypeError* is raised if *s* were incorrectly padded or if there are non-alphabet characters present in the string.

The legacy interface:

```
base64. **decode**(input, output)
```
Decode the contents of the *input* file and write the resulting binary data to the *output* file. *input* and *output* must either be file objects or objects that mimic the file object interface. *input* will be read until *input.read()* returns an empty string.

**base64. decodestring(s)**

Decode the string s, which must contain one or more lines of base64 encoded data, and return a string containing the resulting binary data.

**base64. encode(input, output)**

Encode the contents of the *input* file and write the resulting base64 encoded data to the *output* file. *input* and *output* must either be file objects or objects that mimic the file object interface. *input* will be read until *input.read()* returns an empty string. *encode()* returns the encoded data plus a trailing newline character (\n').

**base64. encodestring(s)**

Encode the string s, which can contain arbitrary binary data, and return a string containing one or more lines of base64-encoded data. *encodestring()* returns a string containing one or more lines of base64-encoded data always including an extra trailing newline (\n').

An example usage of the module:

```python
>>> import base64
>>> encoded = base64.b64encode('data to be encoded')
>>> encoded
'ZGF0YSB0byBiZSBiZSB1bmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
'data to be encoded'
```

**See also:**
Module `binascii`

Support module containing ASCII-to-binary and binary-to-ASCII conversions.

**RFC 1521 - MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies**

Section 5.2, “Base64 Content-Transfer-Encoding,” provides the definition of the base64 encoding.
18.13. **binhex** — Encode and decode binhex4 files

This module encodes and decodes files in binhex4 format, a format allowing representation of Macintosh files in ASCII. On the Macintosh, both forks of a file and the finder information are encoded (or decoded), on other platforms only the data fork is handled.

**Note:** In Python 3.x, special Macintosh support has been removed.

The `binhex` module defines the following functions:

**binhex.** `binhex(input, output)`

Convert a binary file with filename `input` to binhex file `output`. The `output` parameter can either be a filename or a file-like object (any object supporting a `write()` and `close()` method).

**binhex.** `hexbin(input[, output])`

Decode a binhex file `input`. `input` may be a filename or a file-like object supporting `read()` and `close()` methods. The resulting file is written to a file named `output`, unless the argument is omitted in which case the output filename is read from the binhex file.

The following exception is also defined:

**exception** `binhex.Error`

Exception raised when something can’t be encoded using the binhex format (for example, a filename is too long to fit in the filename field), or when input is not properly encoded binhex data.
See also:

**Module binascii**

Support module containing ASCII-to-binary and binary-to-ASCII conversions.
18.13.1. Notes

There is an alternative, more powerful interface to the coder and decoder, see the source for details.

If you code or decode textfiles on non-Macintosh platforms they will still use the old Macintosh newline convention (carriage-return as end of line).

As of this writing, `hexbin()` appears to not work in all cases.
18.14. binascii — Convert between binary and ASCII

The binascii module contains a number of methods to convert between binary and various ASCII-encoded binary representations. Normally, you will not use these functions directly but use wrapper modules like uu, base64, or binhex instead. The binascii module contains low-level functions written in C for greater speed that are used by the higher-level modules.

The binascii module defines the following functions:

binascii.a2b_uu(string)
Convert a single line of uuencoded data back to binary and return the binary data. Lines normally contain 45 (binary) bytes, except for the last line. Line data may be followed by whitespace.

binascii.b2a_uu(data)
Convert binary data to a line of ASCII characters, the return value is the converted line, including a newline char. The length of data should be at most 45.

binascii.a2b_base64(string)
Convert a block of base64 data back to binary and return the binary data. More than one line may be passed at a time.

binascii.b2a_base64(data)
Convert binary data to a line of ASCII characters in base64 coding. The return value is the converted line, including a newline char. The newline is added because the original use case for this function was to feed it a series of 57 byte input lines to get output lines that conform to the MIME-base64 standard. Otherwise the output conforms to RFC 3548.
binascii.a2b_qp(string[, header])

Convert a block of quoted-printable data back to binary and return the binary data. More than one line may be passed at a time. If the optional argument header is present and true, underscores will be decoded as spaces.

binascii.b2a_qp(data[, quotetabs, istext, header])

Convert binary data to a line(s) of ASCII characters in quoted-printable encoding. The return value is the converted line(s). If the optional argument quotetabs is present and true, all tabs and spaces will be encoded. If the optional argument istext is present and true, newlines are not encoded but trailing whitespace will be encoded. If the optional argument header is present and true, spaces will be encoded as underscores per RFC1522. If the optional argument header is present and false, newline characters will be encoded as well; otherwise linefeed conversion might corrupt the binary data stream.

binascii.a2b_hqx(string)

Convert binhex4 formatted ASCII data to binary, without doing RLE-decompression. The string should contain a complete number of binary bytes, or (in case of the last portion of the binhex4 data) have the remaining bits zero.

binascii.rledecode_hqx(data)

Perform RLE-decompression on the data, as per the binhex4 standard. The algorithm uses 0x90 after a byte as a repeat indicator, followed by a count. A count of 0 specifies a byte value of 0x90. The routine returns the decompressed data, unless data input data ends in an orphaned repeat indicator, in which case the Incomplete exception is raised.

binascii.rlecode_hqx(data)

Perform binhex4 style RLE-compression on data and return the
result.

binascii.b2a_hqx(data)
Perform hexbin4 binary-to-ASCII translation and return the resulting string. The argument should already be RLE-coded, and have a length divisible by 3 (except possibly the last fragment).

binascii.crc_hqx(data, crc)
Compute the binhex4 crc value of data, starting with an initial crc and returning the result.

binascii.crc32(data[, crc])
Compute CRC-32, the 32-bit checksum of data, starting with an initial crc. This is consistent with the ZIP file checksum. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm. Use as follows:

```python
print binascii.crc32("hello world")
# Or, in two pieces:
crc = binascii.crc32("hello")
crc = binascii.crc32(" world", crc) & 0xffffffff
print 'crc32 = 0x%08x' % crc
```

**Note:** To generate the same numeric value across all Python versions and platforms use crc32(data) & 0xffffffff. If you are only using the checksum in packed binary format this is not necessary as the return value is the correct 32bit binary representation regardless of sign.

*Changed in version 2.6:* The return value is in the range [-2**31, 2**31-1] regardless of platform. In the past the value would be signed on some platforms and unsigned on others. Use & 0xffffffff on the value if you want it to match Python 3 behavior.

*Changed in version 3.0:* The return value is unsigned and in the range [0, 2**32-1] regardless of platform.
binascii.\texttt{b2a\_hex(data)}

Return the hexadecimal representation of the binary data. Every byte of data is converted into the corresponding 2-digit hex representation. The resulting string is therefore twice as long as the length of data.

\begin{verbatim}
binascii.a2b\_hex(hexstr)
binascii.unhexlify(hexstr)
\end{verbatim}

Return the binary data represented by the hexadecimal string hexstr. This function is the inverse of \texttt{b2a\_hex()}. hexstr must contain an even number of hexadecimal digits (which can be upper or lower case), otherwise a \texttt{TypeError} is raised.

\textit{exception} binascii.\texttt{Error}

Exception raised on errors. These are usually programming errors.

\textit{exception} binascii.\texttt{Incomplete}

Exception raised on incomplete data. These are usually not programming errors, but may be handled by reading a little more data and trying again.

\begin{itemize}
  \item \textbf{See also:}
  \begin{itemize}
    \item \textbf{Module base64}
      Support for RFC compliant base64-style encoding in base 16, 32, and 64.
    \item \textbf{Module binhex}
      Support for the binhex format used on the Macintosh.
    \item \textbf{Module uu}
      Support for UU encoding used on Unix.
    \item \textbf{Module quopri}
      Support for quoted-printable encoding used in MIME email
  \end{itemize}
\end{itemize}
messages.
This module performs quoted-printable transport encoding and decoding, as defined in RFC 1521: “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies”. The quoted-printable encoding is designed for data where there are relatively few nonprintable characters; the base64 encoding scheme available via the base64 module is more compact if there are many such characters, as when sending a graphics file.

**quopri.decode(input, output[, header])**

Decode the contents of the `input` file and write the resulting decoded binary data to the `output` file. `input` and `output` must either be file objects or objects that mimic the file object interface. `input` will be read until `input.readline()` returns an empty string. If the optional argument `header` is present and true, underscore will be decoded as space. This is used to decode “Q”-encoded headers as described in RFC 1522: “MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text”.

**quopri.encode(input, output, quotetabs)**

Encode the contents of the `input` file and write the resulting quoted-printable data to the `output` file. `input` and `output` must either be file objects or objects that mimic the file object interface. `input` will be read until `input.readline()` returns an empty string. `quotetabs` is a flag which controls whether to encode embedded
spaces and tabs; when true it encodes such embedded whitespace, and when false it leaves them unencoded. Note that spaces and tabs appearing at the end of lines are always encoded, as per RFC 1521.

```python
quopri.decodestring(s[, header])
```
Like `decode()` except that it accepts a source string and returns the corresponding decoded string.

```python
quopri.encodestring(s[, quotetabs])
```
Like `encode()` except that it accepts a source string and returns the corresponding encoded string. `quotetabs` is optional (defaulting to 0), and is passed straight through to `encode()`.

See also:

**Module `mimify`**
General utilities for processing of MIME messages.

**Module `base64`**
Encode and decode MIME base64 data
18.16. `uu` — Encode and decode uuencode files

**Source code:** Lib/uu.py

This module encodes and decodes files in uuencode format, allowing arbitrary binary data to be transferred over ASCII-only connections. Wherever a file argument is expected, the methods accept a file-like object. For backwards compatibility, a string containing a pathname is also accepted, and the corresponding file will be opened for reading and writing; the pathname `'-` is understood to mean the standard input or output. However, this interface is deprecated; it’s better for the caller to open the file itself, and be sure that, when required, the mode is `'rb'` or `'wb'` on Windows.

This code was contributed by Lance Ellinghouse, and modified by Jack Jansen.

The `uu` module defines the following functions:

- `uu.encode(in_file, out_file[, name[, mode]])`
  
  Uuencode file `in_file` into file `out_file`. The uuencoded file will have the header specifying `name` and `mode` as the defaults for the results of decoding the file. The default defaults are taken from `in_file`, or `'-` and `0666` respectively.

- `uu.decode(in_file[, out_file[, mode[, quiet]]])`
  
  This call decodes uuencoded file `in_file` placing the result on file `out_file`. If `out_file` is a pathname, `mode` is used to set the permission bits if the file must be created. Defaults for `out_file`
and *mode* are taken from the uuencode header. However, if the file specified in the header already exists, a *uu.Error* is raised.

`decode()` may print a warning to standard error if the input was produced by an incorrect uuencoder and Python could recover from that error. Setting `quiet` to a true value silences this warning.

*exception* `uu.Error`

Subclass of Exception, this can be raised by `uu.decode()` under various situations, such as described above, but also including a badly formatted header, or truncated input file.

**See also:**

**Module** `binascii`

Support module containing ASCII-to-binary and binary-to-ASCII conversions.
Python supports a variety of modules to work with various forms of structured data markup. This includes modules to work with the Standard Generalized Markup Language (SGML) and the Hypertext Markup Language (HTML), and several interfaces for working with the Extensible Markup Language (XML).

It is important to note that modules in the `xml` package require that there be at least one SAX-compliant XML parser available. Starting with Python 2.3, the Expat parser is included with Python, so the `xml.parsers.expat` module will always be available. You may still want to be aware of the PyXML add-on package; that package provides an extended set of XML libraries for Python.

The documentation for the `xml.dom` and `xml.sax` packages are the definition of the Python bindings for the DOM and SAX interfaces.

- **19.1. `HTMLParser`** — Simple HTML and XHTML parser
  - 19.1.1. Example HTML Parser Application
  - 19.1.2. `HTMLParser` Methods
  - 19.1.3. Examples
- **19.2. `sgmllib`** — Simple SGML parser
- **19.3. `htmllib`** — A parser for HTML documents
  - 19.3.1. `HTMLParser` Objects
- **19.4. `htmlentitydefs`** — Definitions of HTML general entities
- **19.5. XML Processing Modules**
- **19.6. XML vulnerabilities**
  - 19.6.1. defused packages
- **19.7. `xml.etree.ElementTree`** — The ElementTree XML API
  - 19.7.1. Tutorial
19.7.1.1. XML tree and elements
19.7.1.2. Parsing XML
19.7.1.3. Finding interesting elements
19.7.1.4. Modifying an XML File
19.7.1.5. Building XML documents
19.7.1.6. Parsing XML with Namespaces
19.7.1.7. Additional resources

19.7.2. XPath support
  19.7.2.1. Example
  19.7.2.2. Supported XPath syntax

19.7.3. Reference
  19.7.3.1. Functions
  19.7.3.2. Element Objects
  19.7.3.3. ElementTree Objects
  19.7.3.4. QName Objects
  19.7.3.5. TreeBuilder Objects
  19.7.3.6. XMLParser Objects

19.8. xml.dom — The Document Object Model API
  19.8.1. Module Contents
  19.8.2. Objects in the DOM
    19.8.2.1. DOMImplementation Objects
    19.8.2.2. Node Objects
    19.8.2.3. NodeList Objects
    19.8.2.4. DocumentType Objects
    19.8.2.5. Document Objects
    19.8.2.6. Element Objects
    19.8.2.7. Attr Objects
    19.8.2.8. NamedNodeMap Objects
    19.8.2.9. Comment Objects
    19.8.2.10. Text and CDATASection Objects
    19.8.2.11. ProcessingInstruction Objects
    19.8.2.12. Exceptions
  19.8.3. Conformance
    19.8.3.1. Type Mapping
19.8.3.2. Accessor Methods

- 19.9. `xml.dom.minidom` — Minimal DOM implementation
  - 19.9.1. DOM Objects
  - 19.9.2. DOM Example
  - 19.9.3. minidom and the DOM standard
- 19.10. `xml.dom.pulldom` — Support for building partial DOM trees
  - 19.10.1. DOMEventStream Objects
- 19.11. `xml.sax` — Support for SAX2 parsers
  - 19.11.1. SAXException Objects
- 19.12. `xml.sax.handler` — Base classes for SAX handlers
  - 19.12.1. ContentHandler Objects
  - 19.12.2. DTDHandler Objects
  - 19.12.3. EntityResolver Objects
  - 19.12.4. ErrorHandler Objects
- 19.13. `xml.sax.saxutils` — SAX Utilities
  - 19.14.1. XMLReader Objects
  - 19.14.2. IncrementalParser Objects
  - 19.14.3. Locator Objects
  - 19.14.4. InputSource Objects
  - 19.14.5. The `Attributes` Interface
  - 19.14.6. The `AttributesNS` Interface
- 19.15. `xml.parsers.expat` — Fast XML parsing using Expat
  - 19.15.1. XMLParser Objects
  - 19.15.2. ExpatError Exceptions
  - 19.15.3. Example
  - 19.15.4. Content Model Descriptions
  - 19.15.5. Expat error constants
19.1. HTMLParser — Simple HTML and XHTML parser

Note: The HTMLParser module has been renamed to html.parser in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

New in version 2.2.

Source code: Lib/HTMLParser.py

This module defines a class HTMLParser which serves as the basis for parsing text files formatted in HTML (HyperText Mark-up Language) and XHTML. Unlike the parser in htmllib, this parser is not based on the SGML parser in sgmllib.

class HTMLParser, HTMLParser

An HTMLParser instance is fed HTML data and calls handler methods when start tags, end tags, text, comments, and other markup elements are encountered. The user should subclass HTMLParser and override its methods to implement the desired behavior.

The HTMLParser class is instantiated without arguments.

Unlike the parser in htmllib, this parser does not check that end tags match start tags or call the end-tag handler for elements which are closed implicitly by closing an outer element.

An exception is defined as well:

definition exception HTMLParser, HTMLParseError
**HTMLParser** is able to handle broken markup, but in some cases it might raise this exception when it encounters an error while parsing. This exception provides three attributes: `msg` is a brief message explaining the error, `lineno` is the number of the line on which the broken construct was detected, and `offset` is the number of characters into the line at which the construct starts.
19.1.1. Example HTML Parser Application

As a basic example, below is a simple HTML parser that uses the `HTMLParser` class to print out start tags, end tags and data as they are encountered:

```python
from HTMLParser import HTMLParser

# create a subclass and override the handler methods
class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print "Encountered a start tag:" , tag

    def handle_endtag(self, tag):
        print "Encountered an end tag:" , tag

    def handle_data(self, data):
        print "Encountered some data:" , data

# instantiate the parser and fed it some HTML
parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>
<body><h1>Parse me!</h1></body></html>')</n```

The output will then be:

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data : Parse me!
Encountered an end tag : h1
Encountered an end tag : body
Encountered an end tag : html
```
19.1.2. HTMLParser Methods

**HTMLParser** instances have the following methods:

**HTMLParser.** `feed(data)`
Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is buffered until more data is fed or `close()` is called. `data` can be either `unicode` or `str`, but passing `unicode` is advised.

**HTMLParser.** `close()`
Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call the `HTMLParser` base class method `close()`.

**HTMLParser.** `reset()`
Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.

**HTMLParser.** `getpos()`
Return current line number and offset.

**HTMLParser.** `get_starttag_text()`
Return the text of the most recently opened start tag. This should not normally be needed for structured processing, but may be useful in dealing with HTML “as deployed” or for re-generating input with minimal changes (whitespace between attributes can be preserved, etc.).

The following methods are called when data or markup elements are encountered and they are meant to be overridden in a subclass. The
base class implementations do nothing (except for handle_startendtag()):

```
HTTPParser.handle_starttag(tag, attrs)
```

This method is called to handle the start of a tag (e.g. `<div id="main">`).

The *tag* argument is the name of the tag converted to lower case. The *attrs* argument is a list of `(name, value)` pairs containing the attributes found inside the tag’s `<>` brackets. The *name* will be translated to lower case, and quotes in the *value* have been removed, and character and entity references have been replaced.

For instance, for the tag `<A HREF="https://www.cwi.nl/">`, this method would be called as `handle_starttag('a', [('href', 'https://www.cwi.nl/')])`.

*Changed in version 2.6:* All entity references from `htmlentitydefs` are now replaced in the attribute values.

```
HTTPParser.handle_endtag(tag)
```

This method is called to handle the end tag of an element (e.g. `</div>`).

The *tag* argument is the name of the tag converted to lower case.

```
HTTPParser.handle_startendtag(tag, attrs)
```

Similar to `handle_starttag()`, but called when the parser encounters an XHTML-style empty tag (`<img ... />`). This method may be overridden by subclasses which require this particular lexical information; the default implementation simply calls `handle_starttag()` and `handle_endtag()`.
HTMLParser.handle_data(data)

This method is called to process arbitrary data (e.g. text nodes and the content of `<script>...</script>` and `<style>...</style>`).

HTMLParser.handle_entityref(name)

This method is called to process a named character reference of the form `&name;` (e.g. `&gt;`), where name is a general entity reference (e.g. 'gt').

HTMLParser.handle_charref(name)

This method is called to process decimal and hexadecimal numeric character references of the form `&#NNN;` and `&#xNNN;`. For example, the decimal equivalent for `&gt;` is `&#62;`, whereas the hexadecimal is `&#x3E;`; in this case the method will receive '62' or 'x3E'.

HTMLParser.handle_comment(data)

This method is called when a comment is encountered (e.g. `<!-- comment--->`).

For example, the comment `<!-- comment -->` will cause this method to be called with the argument 'comment'.

The content of Internet Explorer conditional comments (condcoms) will also be sent to this method, so, for `<!--[if IE 9]>IE9-specific content<![endif]-->`, this method will receive '[if IE 9]>IE9-specific content<![endif]'.

HTMLParser.handle_decl(decl)

This method is called to handle an HTML doctype declaration (e.g. `<!DOCTYPE html>`).

The decl parameter will be the entire contents of the declaration
inside the `<!....>` markup (e.g. `'DOCTYPE html'`).

**HTMLParser.handle_pi(data)**

This method is called when a processing instruction is encountered. The *data* parameter will contain the entire processing instruction. For example, for the processing instruction `<?proc color='red'>`, this method would be called as `handle_pi("proc color='red'")`.

**Note:** The **HTMLParser** class uses the SGML syntactic rules for processing instructions. An XHTML processing instruction using the trailing '?' will cause the '?' to be included in *data*.

**HTMLParser.unknown_decl(data)**

This method is called when an unrecognized declaration is read by the parser.

The *data* parameter will be the entire contents of the declaration inside the `<![...]>` markup. It is sometimes useful to be overridden by a derived class.
19.1.3. Examples

The following class implements a parser that will be used to illustrate more examples:

```python
from HTMLParser import HTMLParser
from htmlentitydefs import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print "Start tag:", tag
        for attr in attrs:
            print " attr: ", attr

    def handle_endtag(self, tag):
        print "End tag ":, tag

    def handle_data(self, data):
        print "Data ":, data

    def handle_comment(self, data):
        print "Comment ":, data

    def handle_entityref(self, name):
        c = unichr(name2codepoint[name])
        print "Named ent:", c

    def handle_charref(self, name):
        if name.startswith('x'):
            c = unichr(int(name[1:], 16))
        else:
            c = unichr(int(name))
        print "Num ent ":, c

    def handle_decl(self, data):
        print "Decl ":, data

parser = MyHTMLParser()

Parsing a doctype:

>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//')
```
Parsing an element with a few attributes and a title:

```python
gt parser.feed('<img src="python-logo.png" alt="The Python logo">')
Start tag: img
  attr: ('src', 'python-logo.png')
  attr: ('alt', 'The Python logo')
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
  Data: Python
  End tag: h1
```

The content of `<script>` and `<style>` elements is returned as is, without further parsing:

```python
gt parser.feed('<style type="text/css">#python { color: green }</style>
Start tag: style
  attr: ('type', 'text/css')
  Data: #python { color: green }
  End tag: style
```

Parsing comments:

```python
gt parser.feed('<!-- a comment -->
... <!--[if IE 9]>IE-specific content<![endif]-->')
Comment: a comment
Comment: [if IE 9]>IE-specific content<![endif]
```
them to the correct char (note: these 3 references are all equivalent to '>

```python
>>> parser.feed('&gt;&amp;#62;&amp;#x3E;')
Named ent: >
Num ent : >
Num ent : >
```

Feeding incomplete chunks to `feed()` works, but `handle_data()` might be called more than once:

```python
>>> for chunk in ['<sp', 'an>buff', 'ered ', 'text</s', 'pan>']
...    parser.feed(chunk)
...  Start tag: span
  Data : buff
  Data : ered
  Data : text
  End tag : span
```

Parsing invalid HTML (e.g. unquoted attributes) also works:

```python
>>> parser.feed('<p><a class=link href=#main>tag soup</a></p>')
Start tag: p
Start tag: a
  attr: ('class', 'link')
  attr: ('href', '#main')
Data : tag soup
End tag : p
End tag : a
```
19.2. sgmllib — Simple SGML parser

 Deprecated since version 2.6: The sgmllib module has been removed in Python 3.

This module defines a class `SGMLParser` which serves as the basis for parsing text files formatted in SGML (Standard Generalized Markup Language). In fact, it does not provide a full SGML parser — it only parses SGML insofar as it is used by HTML, and the module only exists as a base for the `htmllib` module. Another HTML parser which supports XHTML and offers a somewhat different interface is available in the `HTMLParser` module.

```python
class sgmllib.SGMLParser
```

The `SGMLParser` class is instantiated without arguments. The parser is hardcoded to recognize the following constructs:

- Opening and closing tags of the form `<tag attr="value" ...>` and `</tag>`, respectively.
- Numeric character references of the form `&#name;`.
- Entity references of the form `&name;`.
- SGML comments of the form `<!--text-->`. Note that spaces, tabs, and newlines are allowed between the trailing `>` and the immediately preceding `-->`.

A single exception is defined as well:

```python
exception sgmllib.SGMLParseError
```

Exception raised by the `SGMLParser` class when it encounters an error while parsing.
New in version 2.1.

*SGMLParser* instances have the following methods:

**SGMLParser.** `.reset()`
Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.

**SGMLParser.** `.setnomoretags()`
Stop processing tags. Treat all following input as literal input (CDATA). (This is only provided so the HTML tag `<PLAINTEXT>` can be implemented.)

**SGMLParser.** `.setliteral()`
Enter literal mode (CDATA mode).

**SGMLParser.** `.feed(data)`
Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is buffered until more data is fed or `.close()` is called.

**SGMLParser.** `.close()`
Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call `.close()`.

**SGMLParser.** `.get_starttag_text()`
Return the text of the most recently opened start tag. This should not normally be needed for structured processing, but may be useful in dealing with HTML “as deployed” or for re-generating input with minimal changes (whitespace between attributes can be preserved, etc.).

**SGMLParser.** `.handle_starttag(tag, method, attributes)`
This method is called to handle start tags for which either a `start_tag()` or `do_tag()` method has been defined. The `tag` argument is the name of the tag converted to lower case, and the `method` argument is the bound method which should be used to support semantic interpretation of the start tag. The `attributes` argument is a list of `(name, value)` pairs containing the attributes found inside the tag’s <> brackets.

The `name` has been translated to lower case. Double quotes and backslashes in the `value` have been interpreted, as well as known character references and known entity references terminated by a semicolon (normally, entity references can be terminated by any non-alphanumerical character, but this would break the very common case of `<A HREF="url?spam=1&eggs=2">` when eggs is a valid entity name).

For instance, for the tag `<A HREF="http://www.cwi.nl/">`, this method would be called as `unknown_starttag('a', [('href', 'http://www.cwi.nl/')]). The base implementation simply calls `method` with `attributes` as the only argument.

*New in version 2.5:* Handling of entity and character references within attribute values.

**SGMLParser.** `handle_endtag(tag, method)`

This method is called to handle endtags for which an `end_tag()` method has been defined. The `tag` argument is the name of the tag converted to lower case, and the `method` argument is the bound method which should be used to support semantic interpretation of the end tag. If no `end_tag()` method is defined for the closing element, this handler is not called. The base implementation simply calls `method`.

**SGMLParser.** `handle_data(data)`
This method is called to process arbitrary data. It is intended to be overridden by a derived class; the base class implementation does nothing.

**SGMLParser.handle_charref***(ref)**

This method is called to process a character reference of the form `&#ref;`. The base implementation uses **convert_charref()** to convert the reference to a string. If that method returns a string, it is passed to **handle_data()**, otherwise **unknown_charref**(ref) is called to handle the error.

*Changed in version 2.5:* Use **convert_charref()** instead of hard-coding the conversion.

**SGMLParser.convert_charref(ref)**

Convert a character reference to a string, or **None**. ref is the reference passed in as a string. In the base implementation, ref must be a decimal number in the range 0–255. It converts the code point found using the **convert_codepoint()** method. If ref is invalid or out of range, this method returns **None**. This method is called by the default **handle_charref()** implementation and by the attribute value parser.

*New in version 2.5.*

**SGMLParser.convert_codepoint**(codepoint)**

Convert a code point to a **str** value. Encodings can be handled here if appropriate, though the rest of **sgmllib** is oblivious on this matter.

*New in version 2.5.*

**SGMLParser.handle_entityref**(ref)**

This method is called to process a general entity reference of the
form &ref; where ref is a general entity reference. It converts ref by passing it to convert_entityref(). If a translation is returned, it calls the method handle_data() with the translation; otherwise, it calls the method unknown_entityref(ref). The default entitydefs defines translations for &amp;, &apos, &gt;, &lt;, and &quot;.

Changed in version 2.5: Use convert_entityref() instead of hard-coding the conversion.

SGMLParser.convert_entityref(ref)
Convert a named entity reference to a str value, or None. The resulting value will not be parsed. ref will be only the name of the entity. The default implementation looks for ref in the instance (or class) variable entitydefs which should be a mapping from entity names to corresponding translations. If no translation is available for ref, this method returns None. This method is called by the default handle_entityref() implementation and by the attribute value parser.

New in version 2.5.

SGMLParser.handle_comment(comment)
This method is called when a comment is encountered. The comment argument is a string containing the text between the <!-- and --> delimiters, but not the delimiters themselves. For example, the comment <!--text--> will cause this method to be called with the argument 'text'. The default method does nothing.

SGMLParser.handle_decl(data)
Method called when an SGML declaration is read by the parser. In practice, the doctype declaration is the only thing observed in
HTML, but the parser does not discriminate among different (or broken) declarations. Internal subsets in a `DOCTYPE` declaration are not supported. The `data` parameter will be the entire contents of the declaration inside the `<!...>` markup. The default implementation does nothing.

SGMLParser. `report_unbalanced(tag)`
This method is called when an end tag is found which does not correspond to any open element.

SGMLParser. `unknown_starttag(tag, attributes)`
This method is called to process an unknown start tag. It is intended to be overridden by a derived class; the base class implementation does nothing.

SGMLParser. `unknown_endtag(tag)`
This method is called to process an unknown end tag. It is intended to be overridden by a derived class; the base class implementation does nothing.

SGMLParser. `unknown_charref(ref)`
This method is called to process unresolvable numeric character references. Refer to `handle_charref()` to determine what is handled by default. It is intended to be overridden by a derived class; the base class implementation does nothing.

SGMLParser. `unknown_entityref(ref)`
This method is called to process an unknown entity reference. It is intended to be overridden by a derived class; the base class implementation does nothing.

Apart from overriding or extending the methods listed above, derived classes may also define methods of the following form to define processing of specific tags. Tag names in the input stream are case independent; the `tag` occurring in method names must be in lower
case:

SGMLParser.\texttt{start\_tag}(\texttt{attributes})

This method is called to process an opening tag \texttt{tag}. It has preference over \texttt{do\_tag()}. The \texttt{attributes} argument has the same meaning as described for \texttt{handle\_starttag()} above.

SGMLParser.\texttt{do\_tag}(\texttt{attributes})

This method is called to process an opening tag \texttt{tag} for which no \texttt{start\_tag()} method is defined. The \texttt{attributes} argument has the same meaning as described for \texttt{handle\_starttag()} above.

SGMLParser.\texttt{end\_tag()}

This method is called to process a closing tag \texttt{tag}.

Note that the parser maintains a stack of open elements for which no end tag has been found yet. Only tags processed by \texttt{start\_tag()} are pushed on this stack. Definition of an \texttt{end\_tag()} method is optional for these tags. For tags processed by \texttt{do\_tag()} or by \texttt{unknown\_tag()}, no \texttt{end\_tag()} method must be defined; if defined, it will not be used. If both \texttt{start\_tag()} and \texttt{do\_tag()} methods exist for a tag, the \texttt{start\_tag()} method takes precedence.
19.3. htmlib — A parser for HTML documents

*Deprecated since version 2.6:* The htmlib module has been removed in Python 3. Use HTMLParser instead in Python 2, and the equivalent, html.parser, in Python 3.

This module defines a class which can serve as a base for parsing text files formatted in the HyperText Mark-up Language (HTML). The class is not directly concerned with I/O — it must be provided with input in string form via a method, and makes calls to methods of a “formatter” object in order to produce output. The HTMLParser class is designed to be used as a base class for other classes in order to add functionality, and allows most of its methods to be extended or overridden. In turn, this class is derived from and extends the SGMLParser class defined in module sgmllib. The HTMLParser implementation supports the HTML 2.0 language as described in RFC 1866. Two implementations of formatter objects are provided in the formatter module; refer to the documentation for that module for information on the formatter interface.

The following is a summary of the interface defined by sgmllib.SGMLParser:

- The interface to feed data to an instance is through the `feed()` method, which takes a string argument. This can be called with as little or as much text at a time as desired; `p.feed(a); p.feed(b)` has the same effect as `p.feed(a+b)`. When the data contains complete HTML markup constructs, these are processed immediately; incomplete constructs are saved in a buffer. To force processing of all unprocessed data, call the
close() method.

For example, to parse the entire contents of a file, use:

```python
parser.feed(open('myfile.html').read())
parser.close()
```

- The interface to define semantics for HTML tags is very simple: derive a class and define methods called `start_tag()`, `end_tag()`, or `do_tag()`. The parser will call these at appropriate moments: `start_tag()` or `do_tag()` is called when an opening tag of the form `<tag ...>` is encountered; `end_tag()` is called when a closing tag of the form `<tag>` is encountered. If an opening tag requires a corresponding closing tag, like `<H1>` ... `</H1>`, the class should define the `start_tag()` method; if a tag requires no closing tag, like `<P>`, the class should define the `do_tag()` method.

The module defines a parser class and an exception:

```python
class htmllib.HTMLParser(formatter)
    This is the basic HTML parser class. It supports all entity names required by the XHTML 1.0 Recommendation (https://www.w3.org/TR/xhtml1). It also defines handlers for all HTML 2.0 and many HTML 3.0 and 3.2 elements.
```

```python
exception htmllib.HTMLParseError
    Exception raised by the HTMLParser class when it encounters an error while parsing.
```

*New in version 2.4.*

**See also:**

**Module formatter**
Interface definition for transforming an abstract flow of formatting events into specific output events on writer objects.

**Module HTMLParser**
Alternate HTML parser that offers a slightly lower-level view of the input, but is designed to work with XHTML, and does not implement some of the SGML syntax not used in “HTML as deployed” and which isn’t legal for XHTML.

**Module htmlentitydefs**
Definition of replacement text for XHTML 1.0 entities.

**Module sgmllib**
Base class for HTMLParser.
19.3.1. HTMLParser Objects

In addition to tag methods, the HTMLParser class provides some additional methods and instance variables for use within tag methods.

HTMLParser. formatter
This is the formatter instance associated with the parser.

HTMLParser. nofill
Boolean flag which should be true when whitespace should not be collapsed, or false when it should be. In general, this should only be true when character data is to be treated as “preformatted” text, as within a <PRE> element. The default value is false. This affects the operation of handle_data() and save_end().

HTMLParser. anchor_bgn(href, name, type)
This method is called at the start of an anchor region. The arguments correspond to the attributes of the <A> tag with the same names. The default implementation maintains a list of hyperlinks (defined by the HREF attribute for <A> tags) within the document. The list of hyperlinks is available as the data attribute anchorlist.

HTMLParser. anchor_end()
This method is called at the end of an anchor region. The default implementation adds a textual footnote marker using an index into the list of hyperlinks created by anchor_bgn().

HTMLParser. handle_image(source, alt[, ismap[, align[, width[, height]]]]]}
This method is called to handle images. The default implementation simply passes the *alt* value to the `handle_data()` method.

**HTMLParser.** `save_bgn()`  

Begins saving character data in a buffer instead of sending it to the formatter object. Retrieve the stored data via `save_end()`. Use of the `save_bgn()` / `save_end()` pair may not be nested.

**HTMLParser.** `save_end()`  

Ends buffering character data and returns all data saved since the preceding call to `save_bgn()`. If the `nofill` flag is false, whitespace is collapsed to single spaces. A call to this method without a preceding call to `save_bgn()` will raise a `TypeError` exception.
19.4. `htmlentitydefs` — Definitions of HTML general entities

**Note:** The `htmlentitydefs` module has been renamed to `html.entities` in Python 3. The `2to3` tool will automatically adapt imports when converting your sources to Python 3.

**Source code:** Lib/htmlentitydefs.py

This module defines three dictionaries, `name2codepoint`, `codepoint2name`, and `entitydefs`. `entitydefs` is used by the `htmllib` module to provide the `entitydefs` attribute of the `HTMLParser` class. The definition provided here contains all the entities defined by XHTML 1.0 that can be handled using simple textual substitution in the Latin-1 character set (ISO-8859-1).

**htmlentitydefs.**`entitydefs`  
A dictionary mapping XHTML 1.0 entity definitions to their replacement text in ISO Latin-1.

**htmlentitydefs.**`name2codepoint`  
A dictionary that maps HTML entity names to the Unicode code points.  

*New in version 2.3.*

**htmlentitydefs.**`codepoint2name`  
A dictionary that maps Unicode code points to HTML entity names.  

*New in version 2.3.*
19.5. XML Processing Modules

Python's interfaces for processing XML are grouped in the `xml` package.

**Warning:** The XML modules are not secure against erroneous or maliciously constructed data. If you need to parse untrusted or unauthenticated data see [XML vulnerabilities](#).

It is important to note that modules in the `xml` package require that there be at least one SAX-compliant XML parser available. The Expat parser is included with Python, so the `xml.parsers.expat` module will always be available.

The documentation for the `xml.dom` and `xml.sax` packages are the definition of the Python bindings for the DOM and SAX interfaces.

The XML handling submodules are:

- `xml.etree.ElementTree`: the ElementTree API, a simple and lightweight XML processor
- `xml.dom`: the DOM API definition
- `xml.dom.minidom`: a minimal DOM implementation
- `xml.dom.pulldom`: support for building partial DOM trees
- `xml.sax`: SAX2 base classes and convenience functions
- `xml.parsers.expat`: the Expat parser binding
19.6. XML vulnerabilities

The XML processing modules are not secure against maliciously constructed data. An attacker can abuse vulnerabilities for e.g. denial of service attacks, to access local files, to generate network connections to other machines, or to or circumvent firewalls. The attacks on XML abuse unfamiliar features like inline DTD (document type definition) with entities.

The following table gives an overview of the known attacks and if the various modules are vulnerable to them.

<table>
<thead>
<tr>
<th>kind</th>
<th>sax</th>
<th>etree</th>
<th>minidom</th>
<th>pulldom</th>
<th>xmlrpc</th>
</tr>
</thead>
<tbody>
<tr>
<td>billion laughs</td>
<td>Vulnerable</td>
<td>Vulnerable</td>
<td>Vulnerable</td>
<td>Vulnerable</td>
<td>Vulnerable</td>
</tr>
<tr>
<td>quadratic blowup</td>
<td>Vulnerable</td>
<td>Vulnerable</td>
<td>Vulnerable</td>
<td>Vulnerable</td>
<td>Vulnerable</td>
</tr>
<tr>
<td>external entity expansion</td>
<td>Vulnerable</td>
<td>Safe (1)</td>
<td>Safe (2)</td>
<td>Vulnerable</td>
<td>Safe</td>
</tr>
<tr>
<td>DTD retrieval</td>
<td>Vulnerable</td>
<td>Safe</td>
<td>Safe</td>
<td>Vulnerable</td>
<td>Safe</td>
</tr>
<tr>
<td>decompression bomb</td>
<td>Safe</td>
<td>Safe</td>
<td>Safe</td>
<td>Safe</td>
<td>Vulnerable</td>
</tr>
</tbody>
</table>

1. `xml.etree.ElementTree` doesn’t expand external entities and raises a ParserError when an entity occurs.
2. `xml.dom.minidom` doesn’t expand external entities and simply returns the unexpanded entity verbatim.
3. `xmlrpclib` doesn’t expand external entities and omits them.

billion laughs / exponential entity expansion

The Billion Laughs attack – also known as exponential entity expansion – uses multiple levels of nested entities. Each entity refers to another entity several times, the final entity definition contains a small string. Eventually the small string is expanded to several gigabytes. The exponential expansion consumes lots of
CPU time, too.

quadratic blowup entity expansion
A quadratic blowup attack is similar to a Billion Laughs attack; it abuses entity expansion, too. Instead of nested entities it repeats one large entity with a couple of thousand chars over and over again. The attack isn’t as efficient as the exponential case but it avoids triggering countermeasures of parsers against heavily nested entities.

external entity expansion
Entity declarations can contain more than just text for replacement. They can also point to external resources by public identifiers or system identifiers. System identifiers are standard URIs or can refer to local files. The XML parser retrieves the resource with e.g. HTTP or FTP requests and embeds the content into the XML document.

DTD retrieval
Some XML libraries like Python’s xml.dom.pulldom retrieve document type definitions from remote or local locations. The feature has similar implications as the external entity expansion issue.

decompression bomb
The issue of decompression bombs (aka ZIP bomb) apply to all XML libraries that can parse compressed XML stream like gzipped HTTP streams or LZMA-ed files. For an attacker it can reduce the amount of transmitted data by three magnitudes or more.

The documentation of defusedxml on PyPI has further information about all known attack vectors with examples and references.
19.6.1. defused packages

These external packages are recommended for any code that parses untrusted XML data.

**defusedxml** is a pure Python package with modified subclasses of all stdlib XML parsers that prevent any potentially malicious operation. The package also ships with example exploits and extended documentation on more XML exploits like xpath injection.

**defusedexpat** provides a modified libexpat and patched replacement **pyexpat** extension module with countermeasures against entity expansion DoS attacks. Defusedexpat still allows a sane and configurable amount of entity expansions. The modifications will be merged into future releases of Python.

The workarounds and modifications are not included in patch releases as they break backward compatibility. After all inline DTD and entity expansion are well-defined XML features.
The `xml.etree.ElementTree` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see [XML vulnerabilities](#).

Each element has a number of properties associated with it:

- a tag which is a string identifying what kind of data this element represents (the element type, in other words).
- a number of attributes, stored in a Python dictionary.
- a text string.
- an optional tail string.
- a number of child elements, stored in a Python sequence

To create an element instance, use the `Element` constructor or the `SubElement()` factory function.

The `ElementTree` class can be used to wrap an element structure, and convert it from and to XML.

A C implementation of this API is available as `xml.etree.cElementTree`.
See http://effbot.org/zone/element-index.htm for tutorials and links to other docs. Fredrik Lundh’s page is also the location of the development version of the xml.etree.ElementTree.

*Changed in version 2.7:* The ElementTree API is updated to 1.3. For more information, see [Introducing ElementTree 1.3](http://effbot.org/zone/element-index.htm).
19.7.1. Tutorial

This is a short tutorial for using `xml.etree.ElementTree` (ET in short). The goal is to demonstrate some of the building blocks and basic concepts of the module.

19.7.1.1. XML tree and elements

XML is an inherently hierarchical data format, and the most natural way to represent it is with a tree. ET has two classes for this purpose—`ElementTree` represents the whole XML document as a tree, and `Element` represents a single node in this tree. Interactions with the whole document (reading and writing to/from files) are usually done on the `ElementTree` level. Interactions with a single XML element and its sub-elements are done on the `Element` level.

19.7.1.2. Parsing XML

We'll be using the following XML document as the sample data for this section:

```xml
<?xml version="1.0"?>
<data>
  <country name="Liechtenstein">
    <rank>1</rank>
    <year>2008</year>
    <gdppc>141100</gdppc>
    <neighbor name="Austria" direction="E"/>
    <neighbor name="Switzerland" direction="W"/>
  </country>
  <country name="Singapore">
    <rank>4</rank>
    <year>2011</year>
    <gdppc>59900</gdppc>
    <neighbor name="Malaysia" direction="N"/>
  </country>
</data>
```
We have a number of ways to import the data. Reading the file from disk:

```python
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

Reading the data from a string:

```python
root = ET.fromstring(country_data_as_string)
```

`fromstring()` parses XML from a string directly into an `Element`, which is the root element of the parsed tree. Other parsing functions may create an `ElementTree`. Check the documentation to be sure.

As an `Element`, `root` has a tag and a dictionary of attributes:

```python
>>> root.tag
'data'
>>> root.attrib
{}
```

It also has children nodes over which we can iterate:

```python
>>> for child in root:
...     print child.tag, child.attrib
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```
Children are nested, and we can access specific child nodes by index:

```python
>>> root[0][1].text
'2008'
```

### 19.7.1.3. Finding interesting elements

**Element** has some useful methods that help iterate recursively over all the sub-tree below it (its children, their children, and so on). For example, **Element.iter()**:  

```python
>>> for neighbor in root.iter('neighbor'):
...     print neighbor.attrib
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

**Element.findall()** finds only elements with a tag which are direct children of the current element. **Element.find()** finds the *first* child with a particular tag, and **Element.text** accesses the element's text content. **Element.get()** accesses the element's attributes:

```python
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print name, rank
...     print
Liechtenstein 1
Singapore 4
Panama 68
```

More sophisticated specification of which elements to look for is possible by using **XPath**.
19.7.1.4. Modifying an XML File

`ElementTree` provides a simple way to build XML documents and write them to files. The `ElementTree.write()` method serves this purpose.

Once created, an `Element` object may be manipulated by directly changing its fields (such as `Element.text`), adding and modifying attributes (`Element.set()` method), as well as adding new children (for example with `Element.append()`).

Let's say we want to add one to each country's rank, and add an `updated` attribute to the rank element:

```python
>>> for rank in root.iter('rank'):
...    new_rank = int(rank.text) + 1
...    rank.text = str(new_rank)
...    rank.set('updated', 'yes')
...>>> tree.write('output.xml')
```

Our XML now looks like this:

```xml
<?xml version="1.0"?>
<data>
      <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
      </country>
      <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
      </country>
      <country name="Panama">
        <rank updated="yes">69</rank>
```
We can remove elements using `Element.remove()`. Let's say we want to remove all countries with a rank higher than 50:

```python
>>> for country in root.findall('country'):
...    rank = int(country.find('rank').text)
...    if rank > 50:
...        root.remove(country)
...>>>
```

Our XML now looks like this:

```xml
<?xml version="1.0"?><data>  
    <country name="Liechtenstein">  
        <rank updated="yes">2</rank>  
        <year>2008</year>  
        <gdppc>141100</gdppc>  
        <neighbor name="Austria" direction="E"/>  
        <neighbor name="Switzerland" direction="W"/>  
    </country>  
    <country name="Singapore">  
        <rank updated="yes">5</rank>  
        <year>2011</year>  
        <gdppc>59900</gdppc>  
        <neighbor name="Malaysia" direction="N"/>  
    </country>  
</data>
```

### 19.7.1.5. Building XML documents

The `SubElement()` function also provides a convenient way to create new sub-elements for a given element:
19.7.1.6. Parsing XML with Namespaces

If the XML input has namespaces, tags and attributes with prefixes in the form `prefix:sometag` get expanded to `{uri}sometag` where the `prefix` is replaced by the full `URI`. Also, if there is a default `namespace`, that full URI gets prepended to all of the non-prefixed tags.

Here is an XML example that incorporates two namespaces, one with the prefix “fictional” and the other serving as the default namespace:

```xml
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
       xmlns="http://people.example.com">
  <actor>
    <name>John Cleese</name>
    <fictional:character>Lancelot</fictional:character>
    <fictional:character>Archie Leach</fictional:character>
  </actor>
  <actor>
    <name>Eric Idle</name>
    <fictional:character>Sir Robin</fictional:character>
    <fictional:character>Gunther</fictional:character>
    <fictional:character>Commander Clement</fictional:character>
  </actor>
</actors>
```

One way to search and explore this XML example is to manually add the URI to every tag or attribute in the xpath of a `find()` or `findall()`:
A better way to search the namespaced XML example is to create a dictionary with your own prefixes and use those in the search functions:

```python	ns = {'real_person': 'http://people.example.com',
       'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print name.text
    for char in actor.findall('role:character', ns):
        print ' |-->', char.text
```

These two approaches both output:

```
John Cleese
    |--> Lancelot
    |--> Archie Leach
Eric Idle
    |--> Sir Robin
    |--> Gunther
    |--> Commander Clement
```

19.7.1.7. Additional resources

See http://effbot.org/zone/element-index.htm for tutorials and links to other docs.
19.7.2. XPath support

This module provides limited support for XPath expressions for locating elements in a tree. The goal is to support a small subset of the abbreviated syntax; a full XPath engine is outside the scope of the module.

19.7.2.1. Example

Here's an example that demonstrates some of the XPath capabilities of the module. We'll be using the countrydata XML document from the Parsing XML section:

```python
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the top elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall("./year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall(".//*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of their parent
root.findall("./neighbor[2]")
```

19.7.2.2. Supported XPath syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selects all child elements with the given tag.</td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>tag</code></td>
<td>For example, <code>spam</code> selects all child elements named <code>spam</code>, and <code>spam/egg</code> selects all grandchildren named <code>egg</code> in all children named <code>spam</code>.</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Selects all child elements. For example, <code>*/egg</code> selects all grandchildren named <code>egg</code>.</td>
</tr>
<tr>
<td><code>.</code></td>
<td>Selects the current node. This is mostly useful at the beginning of the path, to indicate that it's a relative path.</td>
</tr>
<tr>
<td><code>//</code></td>
<td>Selects all subelements, on all levels beneath the current element. For example, <code>../egg</code> selects all <code>egg</code> elements in the entire tree.</td>
</tr>
<tr>
<td><code>..</code></td>
<td>Selects the parent element.</td>
</tr>
<tr>
<td><code>[@attrib]</code></td>
<td>Selects all elements that have the given attribute.</td>
</tr>
<tr>
<td><code>[@attrib='value']</code></td>
<td>Selects all elements for which the given attribute has the given value. The value cannot contain quotes.</td>
</tr>
<tr>
<td><code>[tag]</code></td>
<td>Selects all elements that have a child named <code>tag</code>. Only immediate children are supported.</td>
</tr>
<tr>
<td><code>[tag='text']</code></td>
<td>Selects all elements that have a child named <code>tag</code> whose complete text content, including descendants, equals the given <code>text</code>.</td>
</tr>
<tr>
<td><code>[position]</code></td>
<td>Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression <code>last()</code> (for the last position), or a position relative to the last position (e.g. <code>last()-1</code>).</td>
</tr>
</tbody>
</table>

Predicates (expressions within square brackets) must be preceded by a tag name, an asterisk, or another predicate. `position` predicates must be preceded by a tag name.
19.7.3. Reference

19.7.3.1. Functions

xml.etree.ElementTree.Comment(text=None)
Comment element factory. This factory function creates a special
element that will be serialized as an XML comment by the
standard serializer. The comment string can be either a bytestring
or a Unicode string. text is a string containing the comment string.
Returns an element instance representing a comment.

xml.etree.ElementTree.dump(elem)
Writes an element tree or element structure to sys.stdout. This
function should be used for debugging only.

The exact output format is implementation dependent. In this
version, it's written as an ordinary XML file.

elem is an element tree or an individual element.

xml.etree.ElementTree.fromstring(text)
Parses an XML section from a string constant. Same as XML().
text is a string containing XML data. Returns an Element instance.

xml.etree.ElementTree.fromstringlist(sequence, parser=None)
Parses an XML document from a sequence of string fragments.
sequence is a list or other sequence containing XML data
fragments. parser is an optional parser instance. If not given, the
standard XMLParser parser is used. Returns an Element instance.

New in version 2.7.

xml.etree.ElementTree.iselement(element)
Checks if an object appears to be a valid element object. element
is an element instance. Returns a true value if this is an element object.

```
xml.etree.ElementTree.iterparse(source, events=None, parser=None)
```

Parses an XML section into an element tree incrementally, and reports what's going on to the user. `source` is a filename or file object containing XML data. `events` is a list of events to report back. If omitted, only “end” events are reported. `parser` is an optional parser instance. If not given, the standard `XMLParser` parser is used. `parser` is not supported by `cElementTree`. Returns an iterator providing `(event, elem)` pairs.

**Note:** `iterparse()` only guarantees that it has seen the “>” character of a starting tag when it emits a “start” event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present. If you need a fully populated element, look for “end” events instead.

```
xml.etree.ElementTree.parse(source, parser=None)
```

Parses an XML section into an element tree. `source` is a filename or file object containing XML data. `parser` is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `ElementTree` instance.

```
xml.etree.ElementTree.ProcessingInstruction(target, text=None)
```

PI element factory. This factory function creates a special element that will be serialized as an XML processing instruction. `target` is a string containing the PI target. `text` is a string containing the PI contents, if given. Returns an element instance, representing a processing instruction.
xml.etree.ElementTree.register_namespace(prefix, uri)

Registers a namespace prefix. The registry is global, and any existing mapping for either the given prefix or the namespace URI will be removed. prefix is a namespace prefix. uri is a namespace uri. Tags and attributes in this namespace will be serialized with the given prefix, if at all possible.

New in version 2.7.

xml.etree.ElementTree.SubElement(parent, tag, attrib={}, **extra)

Subelement factory. This function creates an element instance, and appends it to an existing element.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. parent is the parent element. tag is the subelement name. attrib is an optional dictionary, containing element attributes. extra contains additional attributes, given as keyword arguments. Returns an element instance.

xml.etree.ElementTree.tostring(element, encoding="us-ascii", method="xml")

Generates a string representation of an XML element, including all subelements. element is an Element instance. encoding [1] is the output encoding (default is US-ASCII). method is either "xml", "html" or "text" (default is "xml"). Returns an encoded string containing the XML data.

xml.etree.ElementTree.tostringlist(element, encoding="us-ascii", method="xml")

Generates a string representation of an XML element, including all subelements. element is an Element instance. encoding [1] is the output encoding (default is US-ASCII). method is either "xml", "html" or "text" (default is "xml"). Returns a list of encoded strings containing the XML data. It does not guarantee
any specific sequence, except that
"".join(tostringlist(element)) == tostring(element).

New in version 2.7.

xml.etree.ElementTree.XML(text, parser=None)
Parses an XML section from a string constant. This function can be used to embed “XML literals” in Python code. text is a string containing XML data. parser is an optional parser instance. If not given, the standard XMLParser parser is used. Returns an Element instance.

xml.etree.ElementTree.XMLID(text, parser=None)
Parses an XML section from a string constant, and also returns a dictionary which maps from element id:s to elements. text is a string containing XML data. parser is an optional parser instance. If not given, the standard XMLParser parser is used. Returns a tuple containing an Element instance and a dictionary.

19.7.3.2. Element Objects

class xml.etree.ElementTree.Element(tag, attrib={}, **extra)
Element class. This class defines the Element interface, and provides a reference implementation of this interface.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. tag is the element name. attrib is an optional dictionary, containing element attributes. extra contains additional attributes, given as keyword arguments.

tag
A string identifying what kind of data this element represents (the element type, in other words).

text
**tail**

These attributes can be used to hold additional data associated with the element. Their values are usually strings but may be any application-specific object. If the element is created from an XML file, the `text` attribute holds either the text between the element's start tag and its first child or end tag, or `None`, and the `tail` attribute holds either the text between the element's end tag and the next tag, or `None`. For the XML data

```xml
<a><b>1</b><c>2</c><d/></a>
```

the `a` element has `None` for both `text` and `tail` attributes, the `b` element has `text "1"` and `tail "4"`, the `c` element has `text "2"` and `tail None`, and the `d` element has `text None` and `tail "3"`.

To collect the inner text of an element, see `itertext()`, for example `"".join(element.itertext())`.

Applications may store arbitrary objects in these attributes.

**attrib**

A dictionary containing the element's attributes. Note that while the `attrib` value is always a real mutable Python dictionary, an ElementTree implementation may choose to use another internal representation, and create the dictionary only if someone asks for it. To take advantage of such implementations, use the dictionary methods below whenever possible.

The following dictionary-like methods work on the element attributes.

**clear()**

Resets an element. This function removes all subelements,
clears all attributes, and sets the text and tail attributes to None.

**get**(key, default=None)

Gets the element attribute named key.

Returns the attribute value, or default if the attribute was not found.

**items()**

Returns the element attributes as a sequence of (name, value) pairs. The attributes are returned in an arbitrary order.

**keys()**

Returns the elements attribute names as a list. The names are returned in an arbitrary order.

**set**(key, value)

Set the attribute key on the element to value.

The following methods work on the element's children (subelements).

**append**(subelement)

Adds the element subelement to the end of this elements internal list of subelements.

**extend**(subelements)

Appends subelements from a sequence object with zero or more elements. Raises AssertionError if a subelement is not a valid object.

*New in version 2.7.*

**find**(match)
Finds the first subelement matching `match`. `match` may be a tag name or path. Returns an element instance or `None`.

**findall**(*match*)
Finds all matching subelements, by tag name or path. Returns a list containing all matching elements in document order.

**findtext**(*match*, *default=None*)
Finds text for the first subelement matching `match`. `match` may be a tag name or path. Returns the text content of the first matching element, or *default* if no element was found. Note that if the matching element has no text content an empty string is returned.

**getchildren**()

>`Deprecated since version 2.7: Use list(elem) or iteration.`

**getiterator**(*tag=None*)

>`Deprecated since version 2.7: Use method Element.iter() instead.`

**insert**(*index*, *element*)
Inserts a subelement at the given position in this element.

**iter**(*tag=None*)
Creates a tree *iterator* with the current element as the root. The iterator iterates over this element and all elements below it, in document (depth first) order. If `tag` is not `None` or `'*'`, only elements whose tag equals *tag* are returned from the iterator. If the tree structure is modified during iteration, the result is undefined.

`New in version 2.7.`
iterfind(*match*)
Finds all matching subelements, by tag name or path. Returns an iterable yielding all matching elements in document order.

*New in version 2.7.*

itertext()
Creates a text iterator. The iterator loops over this element and all subelements, in document order, and returns all inner text.

*New in version 2.7.*

makeelement(*tag*, *attrib*)
Creates a new element object of the same type as this element. Do not call this method, use the SubElement() factory function instead.

remove(*subelement*)
Removes subelement from the element. Unlike the find* methods this method compares elements based on the instance identity, not on tag value or contents.

Element objects also support the following sequence type methods for working with subelements: __delitem__(), __getitem__(), __setitem__(), __len__().

Caution: Elements with no subelements will test as **False**. This behavior will change in future versions. Use specific `len(elem)` or `elem is None` test instead.

element = root.find('foo')
if not element:  # careful!
    print "element not found, or element has no subelements"
class xml.etree.ElementTree.**ElementTree**(element=None, file=None)

ElementTree wrapper class. This class represents an entire element hierarchy, and adds some extra support for serialization to and from standard XML.

`element` is the root element. The tree is initialized with the contents of the XML `file` if given.

_**setroot**(element)

Replaces the root element for this tree. This discards the current contents of the tree, and replaces it with the given element. Use with care. `element` is an element instance.

**find**(match)

Same as `Element.find()`, starting at the root of the tree.

**findall**(match)

Same as `Element.findall()`, starting at the root of the tree.

**findtext**(match, default=None)

Same as `Element.findtext()`, starting at the root of the tree.

**getiterator**(tag=None)

*Deprecated since version 2.7: Use method `ElementTree.iter()` instead.*

**getroot**()

Returns the root element for this tree.
iter(tag=None)

Creates and returns a tree iterator for the root element. The iterator loops over all elements in this tree, in section order. tag is the tag to look for (default is to return all elements).

iterfind(match)

Finds all matching subelements, by tag name or path. Same as getroot().iterfind(match). Returns an iterable yielding all matching elements in document order.

New in version 2.7.

parse(source, parser=None)

Loads an external XML section into this element tree. source is a file name or file object. parser is an optional parser instance. If not given, the standard XMLParser parser is used. Returns the section root element.

write(file, encoding="us-ascii", xml_declaration=None, default_namespace=None, method="xml")

Writes the element tree to a file, as XML. file is a file name, or a file object opened for writing. encoding [1] is the output encoding (default is US-ASCII). xml_declaration controls if an XML declaration should be added to the file. Use False for never, True for always, None for only if not US-ASCII or UTF-8 (default is None). default_namespace sets the default XML namespace (for “xmlns”). method is either "xml", "html" or "text" (default is "xml"). Returns an encoded string.

This is the XML file that is going to be manipulated:

```xml
<html>
  <head>
    <title>Example page</title>
  </head>
  <body>
```
Example of changing the attribute “target” of every link in first paragraph:

```python
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>

>>> p = tree.find("body/p")  # Finds first occurrence of tag

>>> links = list(p.iter("a"))  # Returns list of all links

>>> for i in links:  # Iterates through all found links
    ...    i.attrib["target"] = "blank"

>>> tree.write("output.xhtml")
```

19.7.3.4. QName Objects

```python
class xml.etree.ElementTree.QName(text_or_uri, tag=None)
```

QName wrapper. This can be used to wrap a QName attribute value, in order to get proper namespace handling on output. `text_or_uri` is a string containing the QName value, in the form `{uri}local`, or, if the tag argument is given, the URI part of a QName. If `tag` is given, the first argument is interpreted as a URI, and this argument is interpreted as a local name. `QName` instances are opaque.

19.7.3.5. TreeBuilder Objects

```python
class xml.etree.ElementTree.TreeBuilder(element_factory=None)
```
Generic element structure builder. This builder converts a sequence of start, data, and end method calls to a well-formed element structure. You can use this class to build an element structure using a custom XML parser, or a parser for some other XML-like format. The `element_factory` is called to create new `Element` instances when given.

**close()**
Flushes the builder buffers, and returns the toplevel document element. Returns an `Element` instance.

**data(data)**
Adds text to the current element. `data` is a string. This should be either a bytestring, or a Unicode string.

**end(tag)**
Closes the current element. `tag` is the element name. Returns the closed element.

**start(tag, attrs)**
Opens a new element. `tag` is the element name. `attrs` is a dictionary containing element attributes. Returns the opened element.

In addition, a custom `TreeBuilder` object can provide the following method:

**doctype(name, pubid, system)**
Handles a doctype declaration. `name` is the doctype name. `pubid` is the public identifier. `system` is the system identifier. This method does not exist on the default `TreeBuilder` class.

*New in version 2.7.*

19.7.3.6. XMLParser Objects
class xml.etree.ElementTree.**XMLParser**(html=0, target=None, encoding=None)

Element structure builder for XML source data, based on the expat parser. *html* are predefined HTML entities. This flag is not supported by the current implementation. *target* is the target object. If omitted, the builder uses an instance of the standard TreeBuilder class. *encoding* [1] is optional. If given, the value overrides the encoding specified in the XML file.

**close()**

Finishes feeding data to the parser. Returns an element structure.

**doctype**(name, pubid, system)

*Deprecated since version 2.7:* Define the TreeBuilder.doctype() method on a custom TreeBuilder target.

**feed**(data)

Feeds data to the parser. *data* is encoded data.

**XMLParser.feed()** calls *target*’s start() method for each opening tag, its end() method for each closing tag, and data is processed by method data(). **XMLParser.close()** calls *target*’s method close(). **XMLParser** can be used not only for building a tree structure. This is an example of counting the maximum depth of an XML file:

```python
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
```
... def end(self, tag): 
    # Called for each closing tag.
    self.depth -= 1

... def data(self, data):
    pass  # We do not need to do anything with data.

... def close(self):  # Called when all data has been parsed.
    return self.maxDepth

>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""
""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""
""""""""""""""""""""""""""""""""""""""
""""""""""""""""""""""""""""""""""""
""""""""""""""""""""""""""
""""""""""""""""""""
""""""""""""""""""
"""""""""""""

>>> parser.feed(exampleXml)
>>> parser.close()
4

Footnotes

[1] The encoding string included in XML output should conform to the appropriate standards. For example, “UTF-8” is valid, but “UTF8” is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.
19.8. xml.dom — The Document Object Model API

New in version 2.0.

The Document Object Model, or “DOM,” is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents. A DOM implementation presents an XML document as a tree structure, or allows client code to build such a structure from scratch. It then gives access to the structure through a set of objects which provided well-known interfaces.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another. If you are looking at a text node, you have no access to a containing element. When you write a SAX application, you need to keep track of your program’s position in the document somewhere in your own code. SAX does not do it for you. Also, if you need to look ahead in the XML document, you are just out of luck.

Some applications are simply impossible in an event driven model with no access to a tree. Of course you could build some sort of tree yourself in SAX events, but the DOM allows you to avoid writing that code. The DOM is a standard tree representation for XML data.

The Document Object Model is being defined by the W3C in stages, or “levels” in their terminology. The Python mapping of the API is substantially based on the DOM Level 2 recommendation.

DOM applications typically start by parsing some XML into a DOM. How this is accomplished is not covered at all by DOM Level 1, and Level 2 provides only limited improvements: There is a
**DOMImplementation** object class which provides access to **Document** creation methods, but no way to access an XML reader/parser/Document builder in an implementation-independent way. There is also no well-defined way to access these methods without an existing **Document** object. In Python, each DOM implementation will provide a function `getDOMImplementation()`. DOM Level 3 adds a Load/Store specification, which defines an interface to the reader, but this is not yet available in the Python standard library.

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification; this portion of the reference manual describes the interpretation of the specification in Python.

The specification provided by the W3C defines the DOM API for Java, ECMAScript, and OMG IDL. The Python mapping defined here is based in large part on the IDL version of the specification, but strict compliance is not required (though implementations are free to support the strict mapping from IDL). See section **Conformance** for a detailed discussion of mapping requirements.

**See also:**

**Document Object Model (DOM) Level 2 Specification**
   The W3C recommendation upon which the Python DOM API is based.

**Document Object Model (DOM) Level 1 Specification**
   The W3C recommendation for the DOM supported by `xml.dom.minidom`.

**Python Language Mapping Specification**
   This specifies the mapping from OMG IDL to Python.
19.8.1. Module Contents

The `xml.dom` contains the following functions:

`xml.dom.registerDOMImplementation(name, factory)`

Register the `factory` function with the name `name`. The factory function should return an object which implements the `DOMImplementation` interface. The factory function can return the same object every time, or a new one for each call, as appropriate for the specific implementation (e.g. if that implementation supports some customization).

`xml.dom.getDOMImplementation([name[, features]])`  
Return a suitable DOM implementation. The `name` is either well-known, the module name of a DOM implementation, or `None`. If it is not `None`, imports the corresponding module and returns a `DOMImplementation` object if the import succeeds. If no name is given, and if the environment variable `PYTHON_DOM` is set, this variable is used to find the implementation.

If name is not given, this examines the available implementations to find one with the required feature set. If no implementation can be found, raise an `ImportError`. The features list must be a sequence of `(feature, version)` pairs which are passed to the `hasFeature()` method on available `DOMImplementation` objects.

Some convenience constants are also provided:

`xml.dom.EMPTY_NAMESPACE`  
The value used to indicate that no namespace is associated with a node in the DOM. This is typically found as the `namespaceURI` of a node, or used as the `namespaceURI` parameter to a
namespaces-specific method.

New in version 2.2.

`xml.dom.XML_NAMESPACE`

The namespace URI associated with the reserved prefix `xml`, as defined by Namespaces in XML (section 4).

New in version 2.2.

`xml.dom.XMLNS_NAMESPACE`

The namespace URI for namespace declarations, as defined by Document Object Model (DOM) Level 2 Core Specification (section 1.1.8).

New in version 2.2.

`xml.dom.XHTML_NAMESPACE`

The URI of the XHTML namespace as defined by XHTML 1.0: The Extensible HyperText Markup Language (section 3.1.1).

New in version 2.2.

In addition, `xml.dom` contains a base `Node` class and the DOM exception classes. The `Node` class provided by this module does not implement any of the methods or attributes defined by the DOM specification; concrete DOM implementations must provide those. The `Node` class provided as part of this module does provide the constants used for the `nodeType` attribute on concrete `Node` objects; they are located within the class rather than at the module level to conform with the DOM specifications.
# 19.8.2. Objects in the DOM

The definitive documentation for the DOM is the DOM specification from the W3C.

Note that DOM attributes may also be manipulated as nodes instead of as simple strings. It is fairly rare that you must do this, however, so this usage is not yet documented.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Section</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMImplementation</td>
<td>DOMImplementation</td>
<td>Interface to the underlying implementation.</td>
</tr>
<tr>
<td>Node</td>
<td>Node Objects</td>
<td>Base interface for most objects in a document.</td>
</tr>
<tr>
<td>NodeList</td>
<td>NodeList Objects</td>
<td>Interface for a sequence of nodes.</td>
</tr>
<tr>
<td>DocumentType</td>
<td>DocumentType Objects</td>
<td>Information about the declarations needed to process a document.</td>
</tr>
<tr>
<td>Document</td>
<td>Document Objects</td>
<td>Object which represents an entire document.</td>
</tr>
<tr>
<td>Element</td>
<td>Element Objects</td>
<td>Element nodes in the document hierarchy.</td>
</tr>
<tr>
<td>Attr</td>
<td>Attr Objects</td>
<td>Attribute value nodes on element nodes.</td>
</tr>
<tr>
<td>Comment</td>
<td>Comment Objects</td>
<td>Representation of comments in the source document.</td>
</tr>
</tbody>
</table>
Text and CDATASection Objects

Nodes containing textual content from the document.

ProcessingInstruction Objects

Processing instruction representation.

An additional section describes the exceptions defined for working with the DOM in Python.

### 19.8.2.1. DOMImplementation Objects

The **DOMImplementation** interface provides a way for applications to determine the availability of particular features in the DOM they are using. DOM Level 2 added the ability to create new **Document** and **DocumentType** objects using the **DOMImplementation** as well.

**DOMImplementation**. `hasFeature(feature, version)`

Return true if the feature identified by the pair of strings *feature* and *version* is implemented.

**DOMImplementation**. `createDocument(namespaceUri, qualifiedName, doctype)`

Return a new **Document** object (the root of the DOM), with a child **Element** object having the given *namespaceUri* and *qualifiedName*. The *doctype* must be a **DocumentType** object created by **createDocumentType()**, or **None**. In the Python DOM API, the first two arguments can also be **None** in order to indicate that no **Element** child is to be created.

**DOMImplementation**. `createDocumentType(qualifiedName, publicId, systemId)`

Return a new **DocumentType** object that encapsulates the given *qualifiedName*, *publicId*, and *systemId* strings, representing the
information contained in an XML document type declaration.

19.8.2.2. Node Objects

All of the components of an XML document are subclasses of \texttt{Node}.

\texttt{Node}. \texttt{nodeType}

An integer representing the node type. Symbolic constants for the types are on the \texttt{Node} object: \texttt{ELEMENT\_NODE}, \texttt{ATTRIBUTE\_NODE}, \texttt{TEXT\_NODE}, \texttt{CDATA\_SECTION\_NODE}, \texttt{ENTITY\_NODE}, \texttt{PROCESSING\_INSTRUCTION\_NODE}, \texttt{COMMENT\_NODE}, \texttt{DOCUMENT\_NODE}, \texttt{DOCUMENT\_TYPE\_NODE}, \texttt{NOTATION\_NODE}. This is a read-only attribute.

\texttt{Node}. \texttt{parentNode}

The parent of the current node, or \texttt{None} for the document node. The value is always a \texttt{Node} object or \texttt{None}. For \texttt{Element} nodes, this will be the parent element, except for the root element, in which case it will be the \texttt{Document} object. For \texttt{Attr} nodes, this is always \texttt{None}. This is a read-only attribute.

\texttt{Node}. \texttt{attributes}

A \texttt{NamedNodeMap} of attribute objects. Only elements have actual values for this; others provide \texttt{None} for this attribute. This is a read-only attribute.

\texttt{Node}. \texttt{previousSibling}

The node that immediately precedes this one with the same parent. For instance the element with an end-tag that comes just before the \texttt{self} element’s start-tag. Of course, XML documents are made up of more than just elements so the previous sibling could be text, a comment, or something else. If this node is the first child of the parent, this attribute will be \texttt{None}. This is a read-only attribute.
Node. **nextSibling**
The node that immediately follows this one with the same parent. See also **previousSibling**. If this is the last child of the parent, this attribute will be None. This is a read-only attribute.

Node. **childNodes**
A list of nodes contained within this node. This is a read-only attribute.

Node. **firstChild**
The first child of the node, if there are any, or None. This is a read-only attribute.

Node. **lastChild**
The last child of the node, if there are any, or None. This is a read-only attribute.

Node. **localName**
The part of the tagName following the colon if there is one, else the entire tagName. The value is a string.

Node. **prefix**
The part of the tagName preceding the colon if there is one, else the empty string. The value is a string, or None.

Node. **namespaceURI**
The namespace associated with the element name. This will be a string or None. This is a read-only attribute.

Node. **nodeName**
This has a different meaning for each node type; see the DOM specification for details. You can always get the information you would get here from another property such as the tagName property for elements or the name property for attributes. For all node types, the value of this attribute will be either a string or
None. This is a read-only attribute.

Node. **nodeValue**

This has a different meaning for each node type; see the DOM specification for details. The situation is similar to that with **nodeName**. The value is a string or None.

Node. **hasAttributes()**

Returns true if the node has any attributes.

Node. **hasChildNodes()**

Returns true if the node has any child nodes.

Node. **isSameNode(other)**

Returns true if other refers to the same node as this node. This is especially useful for DOM implementations which use any sort of proxy architecture (because more than one object can refer to the same node).

**Note:** This is based on a proposed DOM Level 3 API which is still in the “working draft” stage, but this particular interface appears uncontroversial. Changes from the W3C will not necessarily affect this method in the Python DOM interface (though any new W3C API for this would also be supported).

Node. **appendChild(newChild)**

Add a new child node to this node at the end of the list of children, returning newChild. If the node was already in the tree, it is removed first.

Node. **insertBefore(newChild, refChild)**

Insert a new child node before an existing child. It must be the case that refChild is a child of this node; if not, ValueError is raised. newChild is returned. If refChild is None, it inserts newChild at the end of the children’s list.
Node. **removeChild**(oldChild)

Remove a child node. oldChild must be a child of this node; if not, **ValueError** is raised. oldChild is returned on success. If oldChild will not be used further, its **unlink()** method should be called.

Node. **replaceChild**(newChild, oldChild)

Replace an existing node with a new node. It must be the case that oldChild is a child of this node; if not, **ValueError** is raised.

Node. **normalize**()

Join adjacent text nodes so that all stretches of text are stored as single **Text** instances. This simplifies processing text from a DOM tree for many applications.

*New in version 2.1.*

Node. **cloneNode**(deep)

Clone this node. Setting deep means to clone all child nodes as well. This returns the clone.

### 19.8.2.3. NodeList Objects

A **NodeList** represents a sequence of nodes. These objects are used in two ways in the DOM Core recommendation: an **Element** object provides one as its list of child nodes, and the **getElementsByTagName()** and **getElementsByTagNameNS()** methods of **Node** return objects with this interface to represent query results.

The DOM Level 2 recommendation defines one method and one attribute for these objects:

**NodeList.item(i)**

Return the i’th item from the sequence, if there is one, or **None**. The index i is not allowed to be less than zero or greater than or
equal to the length of the sequence.

**NodeList**. `length`

The number of nodes in the sequence.

In addition, the Python DOM interface requires that some additional support is provided to allow `NodeList` objects to be used as Python sequences. All `NodeList` implementations must include support for `__len__()` and `__getitem__()`. This allows iteration over the `NodeList` in `for` statements and proper support for the `len()` built-in function.

If a DOM implementation supports modification of the document, the `NodeList` implementation must also support the `__setitem__()` and `__delitem__()` methods.

### 19.8.2.4. DocumentType Objects

Information about the notations and entities declared by a document (including the external subset if the parser uses it and can provide the information) is available from a `DocumentType` object. The `DocumentType` for a document is available from the `Document` object's `doctype` attribute; if there is no `DOCTYPE` declaration for the document, the document's `doctype` attribute will be set to `None` instead of an instance of this interface.

`DocumentType` is a specialization of `Node`, and adds the following attributes:

`DocumentType. publicId`

The public identifier for the external subset of the document type definition. This will be a string or `None`.

`DocumentType. systemId`

The system identifier for the external subset of the document type
definition. This will be a URI as a string, or None.

DocumentType. **internalSubset**

A string giving the complete internal subset from the document. This does not include the brackets which enclose the subset. If the document has no internal subset, this should be None.

DocumentType. **name**

The name of the root element as given in the DOCTYPE declaration, if present.

DocumentType. **entities**

This is a NamedNodeMap giving the definitions of external entities. For entity names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be None if the information is not provided by the parser, or if no entities are defined.

DocumentType. **notations**

This is a NamedNodeMap giving the definitions of notations. For notation names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be None if the information is not provided by the parser, or if no notations are defined.

**19.8.2.5. Document Objects**

A **Document** represents an entire XML document, including its constituent elements, attributes, processing instructions, comments etc. Remember that it inherits properties from **Node**.

**Document. documentElement**

The one and only root element of the document.

**Document.createElement(tagName)**
Create and return a new element node. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as `insertBefore()` or `appendChild()`.

**Document.** `.createElementNS(namespaceURI, tagName)`

Create and return a new element with a namespace. The `tagName` may have a prefix. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as `insertBefore()` or `appendChild()`.

**Document.** `.createTextNode(data)`

Create and return a text node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

**Document.** `.createComment(data)`

Create and return a comment node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

**Document.** `.createProcessingInstruction(target, data)`

Create and return a processing instruction node containing the `target` and `data` passed as parameters. As with the other creation methods, this one does not insert the node into the tree.

**Document.** `.createAttribute(name)`

Create and return an attribute node. This method does not associate the attribute node with any particular element. You must use `setAttributeNode()` on the appropriate Element object to use the newly created attribute instance.

**Document.** `.createAttributeNS(namespaceURI, qualifiedName)`

Create and return an attribute node with a namespace. The
tagName may have a prefix. This method does not associate the attribute node with any particular element. You must use 
setAttributeNode() on the appropriate Element object to use the newly created attribute instance.

Document. **getElementsByTag**Name**(tagName)**
Search for all descendants (direct children, children’s children, etc.) with a particular element type name.

Document. **getElementsByTag**NameNS**(namespaceURI, localName)**
Search for all descendants (direct children, children’s children, etc.) with a particular namespace URI and localname. The localname is the part of the namespace after the prefix.

### 19.8.2.6. Element Objects

**Element** is a subclass of Node, so inherits all the attributes of that class.

**Element.** **tagName**
The element type name. In a namespace-using document it may have colons in it. The value is a string.

**Element.** **getElementsByTag**Name**(tagName)**
Same as equivalent method in the Document class.

**Element.** **getElementsByTag**NameNS**(namespaceURI, localName)**
Same as equivalent method in the Document class.

**Element.** **hasAttribute**(name)
Returns true if the element has an attribute named by name.

**Element.** **hasAttributeNS**(namespaceURI, localName)
Returns true if the element has an attribute named by namespaceURI and localName.
Element. `getAttribute(name)`

Return the value of the attribute named by `name` as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

Element. `getAttributeNode(attrname)`

Return the Attr node for the attribute named by `attrname`.

Element. `getAttributeNS(namespaceURI, localName)`

Return the value of the attribute named by `namespaceURI` and `localName` as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

Element. `getAttributeNodeNS(namespaceURI, localName)`

Return an attribute value as a node, given a `namespaceURI` and `localName`.

Element. `removeAttribute(name)`

Remove an attribute by name. If there is no matching attribute, a `NotFoundError` is raised.

Element. `removeAttributeNode(oldAttr)`

Remove and return `oldAttr` from the attribute list, if present. If `oldAttr` is not present, a `NotFoundError` is raised.

Element. `removeAttributeNS(namespaceURI, localName)`

Remove an attribute by name. Note that it uses a localName, not a qname. No exception is raised if there is no matching attribute.

Element. `setAttribute(name, value)`

Set an attribute value from a string.

Element. `setAttributeNode(newAttr)`

Add a new attribute node to the element, replacing an existing attribute if necessary if the `name` attribute matches. If a
replacement occurs, the old attribute node will be returned. If newAttr is already in use, InuseAttributeErr will be raised.

Element. `setAttributeNodeNS(newAttr)`
Add a new attribute node to the element, replacing an existing attribute if necessary if the namespaceURI and localName attributes match. If a replacement occurs, the old attribute node will be returned. If newAttr is already in use, InuseAttributeErr will be raised.

Element. `setAttributeNS(namespaceURI, qname, value)`
Set an attribute value from a string, given a namespaceURI and a qname. Note that a qname is the whole attribute name. This is different than above.

### 19.8.2.7. Attr Objects

Attr inherits from Node, so inherits all its attributes.

**Attr. name**
The attribute name. In a namespace-using document it may include a colon.

**Attr. localName**
The part of the name following the colon if there is one, else the entire name. This is a read-only attribute.

**Attr. prefix**
The part of the name preceding the colon if there is one, else the empty string.

**Attr. value**
The text value of the attribute. This is a synonym for the `nodeValue` attribute.
19.8.2.8. NamedNodeMap Objects

NamedNodeMap does not inherit from Node.

NamedNodeMap. length
The length of the attribute list.

NamedNodeMap. item(index)
Return an attribute with a particular index. The order you get the attributes in is arbitrary but will be consistent for the life of a DOM. Each item is an attribute node. Get its value with the value attribute.

There are also experimental methods that give this class more mapping behavior. You can use them or you can use the standardized getAttribute*() family of methods on the Element objects.

19.8.2.9. Comment Objects

Comment represents a comment in the XML document. It is a subclass of Node, but cannot have child nodes.

Comment. data
The content of the comment as a string. The attribute contains all characters between the leading <!-- and trailing -->, but does not include them.

19.8.2.10. Text and CDATASection Objects

The Text interface represents text in the XML document. If the parser and DOM implementation support the DOM's XML extension, portions of the text enclosed in CDATA marked sections are stored in CDATASection objects. These two interfaces are identical, but provide
different values for the `nodeName` attribute.

These interfaces extend the `Node` interface. They cannot have child nodes.

**Text.data**

The content of the text node as a string.

**Note:** The use of a `CDATASection` node does not indicate that the node represents a complete CDATA marked section, only that the content of the node was part of a CDATA section. A single CDATA section may be represented by more than one node in the document tree. There is no way to determine whether two adjacent `CDATASection` nodes represent different CDATA marked sections.

### 19.8.2.11. ProcessingInstruction Objects

Represents a processing instruction in the XML document; this inherits from the `Node` interface and cannot have child nodes.

**ProcessingInstruction.target**

The content of the processing instruction up to the first whitespace character. This is a read-only attribute.

**ProcessingInstruction.data**

The content of the processing instruction following the first whitespace character.

### 19.8.2.12. Exceptions

*New in version 2.1.*

The DOM Level 2 recommendation defines a single exception, `DOMException`, and a number of constants that allow applications to
determine what sort of error occurred. **DOMException** instances carry a `code` attribute that provides the appropriate value for the specific exception.

The Python DOM interface provides the constants, but also expands the set of exceptions so that a specific exception exists for each of the exception codes defined by the DOM. The implementations must raise the appropriate specific exception, each of which carries the appropriate value for the `code` attribute.

*exception* `xml.dom.DOMException`
Base exception class used for all specific DOM exceptions. This exception class cannot be directly instantiated.

*exception* `xml.dom.DomstringSizeErr`
Raised when a specified range of text does not fit into a string. This is not known to be used in the Python DOM implementations, but may be received from DOM implementations not written in Python.

*exception* `xml.dom.HierarchyRequestErr`
Raised when an attempt is made to insert a node where the node type is not allowed.

*exception* `xml.dom.IndexSizeErr`
Raised when an index or size parameter to a method is negative or exceeds the allowed values.

*exception* `xml.dom.InuseAttributeErr`
Raised when an attempt is made to insert an Attr node that is already present elsewhere in the document.

*exception* `xml.dom.InvalidAccessErr`
Raised if a parameter or an operation is not supported on the underlying object.
**exception** xml.dom.**InvalidCharacterErr**

This exception is raised when a string parameter contains a character that is not permitted in the context it’s being used in by the XML 1.0 recommendation. For example, attempting to create an `Element` node with a space in the element type name will cause this error to be raised.

**exception** xml.dom.**InvalidModificationErr**

Raised when an attempt is made to modify the type of a node.

**exception** xml.dom.**InvalidStateErr**

Raised when an attempt is made to use an object that is not defined or is no longer usable.

**exception** xml.dom.**NamespaceErr**

If an attempt is made to change any object in a way that is not permitted with regard to the Namespaces in XML recommendation, this exception is raised.

**exception** xml.dom.**NotFoundError**

Exception when a node does not exist in the referenced context. For example, `NamedNodeMap.removeNamedItem()` will raise this if the node passed in does not exist in the map.

**exception** xml.dom.**UnsupportedErr**

Raised when the implementation does not support the requested type of object or operation.

**exception** xml.dom.**NoDataAllowedErr**

This is raised if data is specified for a node which does not support data.

**exception** xml.dom.**NoModificationAllowedErr**

Raised on attempts to modify an object where modifications are not allowed (such as for read-only nodes).
**exception** xml.dom.*SyntaxErr*

Raised when an invalid or illegal string is specified.

**exception** xml.dom.*WrongDocumentErr*

Raised when a node is inserted in a different document than it currently belongs to, and the implementation does not support migrating the node from one document to the other.

The exception codes defined in the DOM recommendation map to the exceptions described above according to this table:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMSTRING_SIZE_ERR</td>
<td>DomstringSizeErr</td>
</tr>
<tr>
<td>HIERARCHY_REQUEST_ERR</td>
<td>HierarchyRequestErr</td>
</tr>
<tr>
<td>INDEX_SIZE_ERR</td>
<td>IndexSizeErr</td>
</tr>
<tr>
<td>INUSE_ATTRIBUTE_ERR</td>
<td>InuseAttributeErr</td>
</tr>
<tr>
<td>INVALID_ACCESS_ERR</td>
<td>InvalidAccessErr</td>
</tr>
<tr>
<td>INVALID_CHARACTER_ERR</td>
<td>InvalidCharacterErr</td>
</tr>
<tr>
<td>INVALID_MODIFICATION_ERR</td>
<td>InvalidModificationErr</td>
</tr>
<tr>
<td>INVALID_STATE_ERR</td>
<td>InvalidStateErr</td>
</tr>
<tr>
<td>NAMESPACE_ERR</td>
<td>NamespaceErr</td>
</tr>
<tr>
<td>NOT_FOUND_ERR</td>
<td>NotFoundErr</td>
</tr>
<tr>
<td>NOT_SUPPORTED_ERR</td>
<td>NotSupportedErr</td>
</tr>
<tr>
<td>NO_DATA_ALLOWED_ERR</td>
<td>NoDataAllowedErr</td>
</tr>
<tr>
<td>NO_MODIFICATION_ALLOWED_ERR</td>
<td>NoModificationAllowedErr</td>
</tr>
<tr>
<td>SYNTAX_ERR</td>
<td>SyntaxErr</td>
</tr>
<tr>
<td>WRONG_DOCUMENT_ERR</td>
<td>WrongDocumentErr</td>
</tr>
</tbody>
</table>
19.8.3. Conformance

This section describes the conformance requirements and relationships between the Python DOM API, the W3C DOM recommendations, and the OMG IDL mapping for Python.

19.8.3.1. Type Mapping

The primitive IDL types used in the DOM specification are mapped to Python types according to the following table.

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Python Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>IntegerType (with a value of 0 or 1)</td>
</tr>
<tr>
<td>int</td>
<td>IntegerType</td>
</tr>
<tr>
<td>long int</td>
<td>IntegerType</td>
</tr>
<tr>
<td>unsigned int</td>
<td>IntegerType</td>
</tr>
</tbody>
</table>

Additionally, the `DOMString` defined in the recommendation is mapped to a Python string or Unicode string. Applications should be able to handle Unicode whenever a string is returned from the DOM.

The IDL `null` value is mapped to `None`, which may be accepted or provided by the implementation whenever `null` is allowed by the API.

19.8.3.2. Accessor Methods

The mapping from OMG IDL to Python defines accessor functions for IDL `attribute` declarations in much the way the Java mapping does. Mapping the IDL declarations

```plaintext
readonly attribute string someValue;
attribute string anotherValue;
```
yields three accessor functions: a “get” method for someValue (_get_someValue()), and “get” and “set” methods for anotherValue (_get_anotherValue() and _set_anotherValue()). The mapping, in particular, does not require that the IDL attributes are accessible as normal Python attributes: object.someValue is not required to work, and may raise an AttributeError.

The Python DOM API, however, does require that normal attribute access work. This means that the typical surrogates generated by Python IDL compilers are not likely to work, and wrapper objects may be needed on the client if the DOM objects are accessed via CORBA. While this does require some additional consideration for CORBA DOM clients, the implementers with experience using DOM over CORBA from Python do not consider this a problem. Attributes that are declared readonly may not restrict write access in all DOM implementations.

In the Python DOM API, accessor functions are not required. If provided, they should take the form defined by the Python IDL mapping, but these methods are considered unnecessary since the attributes are accessible directly from Python. “Set” accessors should never be provided for readonly attributes.

The IDL definitions do not fully embody the requirements of the W3C DOM API, such as the notion of certain objects, such as the return value of getElementsByTagName(), being “live”. The Python DOM API does not require implementations to enforce such requirements.
19.9. xml.dom.minidom — Minimal DOM implementation

New in version 2.0.

Source code: Lib/xml/dom/minidom.py

xml.dom.minidom is a minimal implementation of the Document Object Model interface, with an API similar to that in other languages. It is intended to be simpler than the full DOM and also significantly smaller. Users who are not already proficient with the DOM should consider using the xml.etree.ElementTree module for their XML processing instead.

Warning: The xml.dom.minidom module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

DOM applications typically start by parsing some XML into a DOM. With xml.dom.minidom, this is done through the parse functions:

```python
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\temp\mydata.xml')  # parse an XML file by name

datasource = open('c:\temp\mydata.xml')
dom2 = parse(datasource)  # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')
```

The parse() function can take either a filename or an open file object.
xml.dom.minidom.\texttt{parse(filename\_or\_file[, parser[, bufsize]])}

Return a \texttt{Document} from the given input. \texttt{filename\_or\_file} may be either a file name, or a file-like object. \texttt{parser}, if given, must be a SAX2 parser object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the \texttt{parseString()} function instead:

xml.dom.minidom.\texttt{parseString(string[, parser])}

Return a \texttt{Document} that represents the \texttt{string}. This method creates a \texttt{StringIO} object for the string and passes that on to \texttt{parse()}.

Both functions return a \texttt{Document} object representing the content of the document.

What the \texttt{parse()} and \texttt{parseString()} functions do is connect an XML parser with a “DOM builder” that can accept parse events from any SAX parser and convert them into a DOM tree. The name of the functions are perhaps misleading, but are easy to grasp when learning the interfaces. The parsing of the document will be completed before these functions return; it’s simply that these functions do not provide a parser implementation themselves.

You can also create a \texttt{Document} by calling a method on a “DOM Implementation” object. You can get this object either by calling the \texttt{getDOMImplementation()} function in the \texttt{xml.dom} package or the \texttt{xml.dom.minidom} module. Using the implementation from the \texttt{xml.dom.minidom} module will always return a \texttt{Document} instance from the minidom implementation, while the version from \texttt{xml.dom} may provide an alternate implementation (this is likely if you have the
PyXML package installed). Once you have a Document, you can add child nodes to it to populate the DOM:

```python
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification. The main property of the document object is the documentElement property. It gives you the main element in the XML document: the one that holds all others. Here is an example program:

```python
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

When you are finished with a DOM tree, you may optionally call the unlink() method to encourage early cleanup of the now-unneeded objects. unlink() is an xml.dom.minidom-specific extension to the DOM API that renders the node and its descendants are essentially useless. Otherwise, Python’s garbage collector will eventually take care of the objects in the tree.

See also:

**Document Object Model (DOM) Level 1 Specification**

The W3C recommendation for the DOM supported by xml.dom.minidom.
19.9.1. DOM Objects

The definition of the DOM API for Python is given as part of the `xml.dom` module documentation. This section lists the differences between the API and `xml.dom.minidom`.

Node. `unlink()`

Break internal references within the DOM so that it will be garbage collected on versions of Python without cyclic GC. Even when cyclic GC is available, using this can make large amounts of memory available sooner, so calling this on DOM objects as soon as they are no longer needed is good practice. This only needs to be called on the `Document` object, but may be called on child nodes to discard children of that node.

Node. `writexml(writer, indent='', addindent='', newl='')`

Write XML to the writer object. The writer should have a `write()` method which matches that of the file object interface. The `indent` parameter is the indentation of the current node. The `addindent` parameter is the incremental indentation to use for subnodes of the current one. The `newl` parameter specifies the string to use to terminate newlines.

For the `Document` node, an additional keyword argument `encoding` can be used to specify the encoding field of the XML header.

*Changed in version 2.1:* The optional keyword parameters `indent`, `addindent`, and `newl` were added to support pretty output.

*Changed in version 2.3:* For the `Document` node, an additional keyword argument `encoding` can be used to specify the encoding field of the XML header.
Node. `toxml([encoding])`

Return the XML that the DOM represents as a string.

With no argument, the XML header does not specify an encoding, and the result is Unicode string if the default encoding cannot represent all characters in the document. Encoding this string in an encoding other than UTF-8 is likely incorrect, since UTF-8 is the default encoding of XML.

With an explicit `encoding` argument, the result is a byte string in the specified encoding. It is recommended that this argument is always specified. To avoid `UnicodeError` exceptions in case of unrepresentable text data, the encoding argument should be specified as “utf-8”.

*Changed in version 2.3:* the `encoding` argument was introduced; see `writexml()`.

Node. `toprettyxml([indent=""", newl=""", encoding=""])`

Return a pretty-printed version of the document. `indent` specifies the indentation string and defaults to a tabulator; `newl` specifies the string emitted at the end of each line and defaults to \n.

*New in version 2.1.*

*Changed in version 2.3:* the encoding argument was introduced; see `writexml()`.

The following standard DOM methods have special considerations with `xml.dom.minidom`:

Node. `cloneNode(deep)`

Although this method was present in the version of `xml.dom.minidom` packaged with Python 2.0, it was seriously
broken. This has been corrected for subsequent releases.
This example program is a fairly realistic example of a simple program. In this particular case, we do not take much advantage of the flexibility of the DOM.

```python
import xml.dom.minidom
document = """
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""
dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print "<html>"
    handleSlideshowTitle(slideshow.getElementsByTagName("title")
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print "</html>"

def handleSlides(slides):
    for slide in slides:
        print "<slide>"
def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print "<title>%s</title>" % getText(title.childNodes)

def handleSlideTitle(title):
    print "<h2>%s</h2>" % getText(title.childNodes)

def handlePoints(points):
    print "<ul>"
    for point in points:
        handlePoint(point)
    print "</ul>"

def handlePoint(point):
    print "<li>%s</li>" % getText(point.childNodes)

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print "<p>%s</p>" % getText(title.childNodes)

handleSlideshow(dom)
19.9.3. minidom and the DOM standard

The `xml.dom.minidom` module is essentially a DOM 1.0-compatible DOM with some DOM 2 features (primarily namespace features).

Usage of the DOM interface in Python is straight-forward. The following mapping rules apply:

- Interfaces are accessed through instance objects. Applications should not instantiate the classes themselves; they should use the creator functions available on the `Document` object. Derived interfaces support all operations (and attributes) from the base interfaces, plus any new operations.
- Operations are used as methods. Since the DOM uses only `in` parameters, the arguments are passed in normal order (from left to right). There are no optional arguments. `void` operations return `None`.
- IDL attributes map to instance attributes. For compatibility with the OMG IDL language mapping for Python, an attribute `foo` can also be accessed through accessor methods `_get_foo()` and `_set_foo()`. `readonly` attributes must not be changed; this is not enforced at runtime.
- The types `short int`, `unsigned int`, `unsigned long long`, and `boolean` all map to Python integer objects.
- The type `DOMString` maps to Python strings. `xml.dom.minidom` supports either byte or Unicode strings, but will normally produce Unicode strings. Values of type `DOMString` may also be `None` where allowed to have the IDL `null` value by the DOM specification from the W3C.
- `const` declarations map to variables in their respective scope (e.g. `xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE`); they must not be changed.
• **DOMException** is currently not supported in `xml.dom.minidom`. Instead, `xml.dom.minidom` uses standard Python exceptions such as `TypeError` and `AttributeError`.

• **NodeList** objects are implemented using Python’s built-in list type. Starting with Python 2.2, these objects provide the interface defined in the DOM specification, but with earlier versions of Python they do not support the official API. They are, however, much more “Pythonic” than the interface defined in the W3C recommendations.

The following interfaces have no implementation in `xml.dom.minidom`:

• **DOMTimeStamp**
• **DocumentType** (added in Python 2.1)
• **DOMImplementation** (added in Python 2.1)
• **CharacterData**
• **CDATASection**
• **Notation**
• **Entity**
• **EntityReference**
• **DocumentFragment**

Most of these reflect information in the XML document that is not of general utility to most DOM users.

**Footnotes**

[1] The encoding string included in XML output should conform to the appropriate standards. For example, “UTF-8” is valid, but “UTF8” is not. See [https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl](https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl) and [https://www.iana.org/assignments/character-sets/character-sets.xhtml](https://www.iana.org/assignments/character-sets/character-sets.xhtml).
19.10. `xml.dom.pulldom` — Support for building partial DOM trees

*New in version 2.0.*

**Source code:** Lib/xml/dom/pulldom.py

`xml.dom.pulldom` allows building only selected portions of a Document Object Model representation of a document from SAX events.

**Warning:** The `xml.dom.pulldom` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see [XML vulnerabilities](#).

```python
class xml.dom.pulldom.PullDOM([documentFactory])
    xml.sax.handler.ContentHandler implementation that ...

class xml.dom.pulldom.DOMEventStream(stream, parser, bufsize)
    ...

class xml.dom.pulldom.SAX2DOM([documentFactory])
    xml.sax.handler.ContentHandler implementation that ...

xml.dom.pulldom.parse(stream_or_string[, parser[, bufsize]])
    ...

xmldom.pulldom.parseString(string[, parser])
    ...

xml.dom.pulldom.default_bufsize
```
Default value for the `bufsize` parameter to `parse()`.

*Changed in version 2.1:* The value of this variable can be changed before calling `parse()` and the new value will take effect.
19.10.1. DOMEventStream Objects

DOMEventStream. `getEvent()`
...

DOMEventStream. `expandNode(node)`
...

DOMEventStream. `reset()`
...
19.11. xml.sax — Support for SAX2 parsers

New in version 2.0.

The xml.sax package provides a number of modules which implement the Simple API for XML (SAX) interface for Python. The package itself provides the SAX exceptions and the convenience functions which will be most used by users of the SAX API.

**Warning:** The xml.sax module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see [XML vulnerabilities](#).

The convenience functions are:

```python
xml.sax.make_parser([parser_list])
```
Create and return a SAX XMLReader object. The first parser found will be used. If parser_list is provided, it must be a sequence of strings which name modules that have a function named create_parser(). Modules listed in parser_list will be used before modules in the default list of parsers.

```python
xml.sax.parse(filename_or_stream, handler[, error_handler])
```
Create a SAX parser and use it to parse a document. The document, passed in as filename_or_stream, can be a filename or a file object. The handler parameter needs to be a SAX ContentHandler instance. If error_handler is given, it must be a SAX ErrorHandler instance; if omitted, SAXParseException will be raised on all errors. There is no return value; all work must be done by the handler passed in.
XML parseString

Similar to parse(), but parses from a buffer string received as a parameter.

A typical SAX application uses three kinds of objects: readers, handlers and input sources. “Reader” in this context is another term for parser, i.e. some piece of code that reads the bytes or characters from the input source, and produces a sequence of events. The events then get distributed to the handler objects, i.e. the reader invokes a method on the handler. A SAX application must therefore obtain a reader object, create or open the input sources, create the handlers, and connect these objects all together. As the final step of preparation, the reader is called to parse the input. During parsing, methods on the handler objects are called based on structural and syntactic events from the input data.

For these objects, only the interfaces are relevant; they are normally not instantiated by the application itself. Since Python does not have an explicit notion of interface, they are formally introduced as classes, but applications may use implementations which do not inherit from the provided classes. The InputSource, Locator, Attributes, AttributesNS, and XMLReader interfaces are defined in the module xml.sax.xmlreader. The handler interfaces are defined in xml.sax.handler. For convenience, InputSource (which is often instantiated directly) and the handler classes are also available from xml.sax. These interfaces are described below.

In addition to these classes, xml.sax provides the following exception classes.

Exception xml.sax.SAXException

Encapsulate an XML error or warning. This class can contain basic error or warning information from either the XML parser or
the application: it can be subclassed to provide additional functionality or to add localization. Note that although the handlers defined in the ErrorHandler interface receive instances of this exception, it is not required to actually raise the exception — it is also useful as a container for information.

When instantiated, msg should be a human-readable description of the error. The optional exception parameter, if given, should be None or an exception that was caught by the parsing code and is being passed along as information.

This is the base class for the other SAX exception classes.

```python
exception xml.sax.SAXParseException(msg, exception, locator)
```
Subclass of SAXException raised on parse errors. Instances of this class are passed to the methods of the SAX ErrorHandler interface to provide information about the parse error. This class supports the SAX Locator interface as well as the SAXException interface.

```python
exception xml.sax.SAXNotRecognizedException(msg[, exception])
```
Subclass of SAXException raised when a SAX XMLReader is confronted with an unrecognized feature or property. SAX applications and extensions may use this class for similar purposes.

```python
exception xml.sax.SAXNotSupportedException(msg[, exception])
```
Subclass of SAXException raised when a SAX XMLReader is asked to enable a feature that is not supported, or to set a property to a value that the implementation does not support. SAX applications and extensions may use this class for similar purposes.

See also:
SAX: The Simple API for XML

This site is the focal point for the definition of the SAX API. It provides a Java implementation and online documentation. Links to implementations and historical information are also available.

**Module `xml.sax.handler`**
Definitions of the interfaces for application-provided objects.

**Module `xml.sax.saxutils`**
Convenience functions for use in SAX applications.

**Module `xml.sax.xmlreader`**
Definitions of the interfaces for parser-provided objects.
19.11.1. SAXException Objects

The `SAXException` exception class supports the following methods:

**SAXException. `getMessage()`**
Return a human-readable message describing the error condition.

**SAXException. `getException()`**
Return an encapsulated exception object, or `None`.
19.12. `xml.sax.handler` — Base classes for SAX handlers

*New in version 2.0.*

The SAX API defines four kinds of handlers: content handlers, DTD handlers, error handlers, and entity resolvers. Applications normally only need to implement those interfaces whose events they are interested in; they can implement the interfaces in a single object or in multiple objects. Handler implementations should inherit from the base classes provided in the module `xml.sax.handler`, so that all methods get default implementations.

```python
class xml.sax.handler.ContentHandler
    This is the main callback interface in SAX, and the one most important to applications. The order of events in this interface mirrors the order of the information in the document.

class xml.sax.handler.DTDHandler
    Handle DTD events.

    This interface specifies only those DTD events required for basic parsing (unparsed entities and attributes).

class xml.sax.handler.EntityResolver
    Basic interface for resolving entities. If you create an object implementing this interface, then register the object with your Parser, the parser will call the method in your object to resolve all external entities.

class xml.sax.handler.ErrorHandler
    Interface used by the parser to present error and warning messages to the application. The methods of this object control whether errors are immediately converted to exceptions or are
handled in some other way.

In addition to these classes, `xml.sax.handler` provides symbolic constants for the feature and property names.

`xml.sax.handler.feature_namespaces`

value: "http://xml.org/sax/features/namespaces"
true: Perform Namespace processing.
false: Optionally do not perform Namespace processing (implies namespace-prefixes; default).
access: (parsing) read-only; (not parsing) read/write

`xml.sax.handler.feature_namespace_prefixes`

value: "http://xml.org/sax/features/namespace-prefixes"
true: Report the original prefixed names and attributes used for Namespace declarations.
false: Do not report attributes used for Namespace declarations, and optionally do not report original prefixed names (default).
access: (parsing) read-only; (not parsing) read/write

`xml.sax.handler.feature_string_interning`

value: "http://xml.org/sax/features/string-interning"
true: All element names, prefixes, attribute names, Namespace URIs, and local names are interned using the built-in intern function.
false: Names are not necessarily interned, although they may be (default).
access: (parsing) read-only; (not parsing) read/write

`xml.sax.handler.feature_validation`

value: "http://xml.org/sax/features/validation"
true: Report all validation errors (implies external-general-entities and external-parameter-entities).
false: Do not report validation errors.
access: (parsing) read-only; (not parsing) read/write
xml.sax.handler.feature_external_ges
value: "http://xml.org/sax/features/external-general-entities"
true: Include all external general (text) entities.
false: Do not include external general entities.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_external_pes
value: "http://xml.org/sax/features/external-parameter-entities"
true: Include all external parameter entities, including the external DTD subset.
false: Do not include any external parameter entities, even the external DTD subset.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.all_features
List of all features.

xml.sax.handler.property_lexical_handler
value: "http://xml.org/sax/properties/lexical-handler"
data type: xml.sax.sax2lib.LexicalHandler (not supported in Python 2)
description: An optional extension handler for lexical events like comments.
access: read/write

xml.sax.handler.property_declaration_handler
value: "http://xml.org/sax/properties/declaration-handler"
data type: xml.sax.sax2lib.DeclHandler (not supported in Python 2)
description: An optional extension handler for DTD-related events other than notations and unparsed entities.
access: read/write
xml.sax.handler. **property_dom_node**

value: "http://xml.org/sax/properties/dom-node"

data type: org.w3c.dom.Node (not supported in Python 2)

description: When parsing, the current DOM node being visited if this is a DOM iterator; when not parsing, the root DOM node for iteration.

access: (parsing) read-only; (not parsing) read/write

xml.sax.handler. **property_xml_string**

value: "http://xml.org/sax/properties/xml-string"

data type: String

description: The literal string of characters that was the source for the current event.

access: read-only

xml.sax.handler. **all_properties**

List of all known property names.
19.12.1. ContentHandler Objects

Users are expected to subclass `ContentHandler` to support their application. The following methods are called by the parser on the appropriate events in the input document:

`ContentHandler.setDocumentLocator(locator)`

Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application’s business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

`ContentHandler.startDocument()`

Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in DTDHandler (except for `setDocumentLocator()`).
ContentHandler. **endDocument()**

Receive notification of the end of a document.

The SAX parser will invoke this method only once, and it will be the last method invoked during the parse. The parser shall not invoke this method until it has either abandoned parsing (because of an unrecoverable error) or reached the end of input.

ContentHandler. **startPrefixMapping**(prefix, uri)

Begin the scope of a prefix-URI Namespace mapping.

The information from this event is not necessary for normal Namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the `feature_namespaces` feature is enabled (the default).

There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the `startPrefixMapping()` and `endPrefixMapping()` events supply the information to the application to expand prefixes in those contexts itself, if necessary.

Note that `startPrefixMapping()` and `endPrefixMapping()` events are not guaranteed to be properly nested relative to each-other: all `startPrefixMapping()` events will occur before the corresponding `startElement()` event, and all `endPrefixMapping()` events will occur after the corresponding `endElement()` event, but their order is not guaranteed.

ContentHandler. **endPrefixMapping**(prefix)

End the scope of a prefix-URI mapping.

See `startPrefixMapping()` for details. This event will always occur after the corresponding `endElement()` event, but the order
of \texttt{endPrefixMapping()} events is not otherwise guaranteed.

\textbf{ContentHandler.} \texttt{startElement(name, attrs)}

Signals the start of an element in non-namespace mode.

The \textit{name} parameter contains the raw XML 1.0 name of the element type as a string and the \textit{attrs} parameter holds an object of the \texttt{Attributes} interface (see \texttt{The Attributes Interface}) containing the attributes of the element. The object passed as \textit{attrs} may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the \texttt{copy()} method of the \textit{attrs} object.

\textbf{ContentHandler.} \texttt{endElement(name)}

Signals the end of an element in non-namespace mode.

The \textit{name} parameter contains the name of the element type, just as with the \texttt{startElement()} event.

\textbf{ContentHandler.} \texttt{startElementNS(name, qname, attrs)}

Signals the start of an element in namespace mode.

The \textit{name} parameter contains the name of the element type as a \texttt{(uri, localname)} tuple, the \textit{qname} parameter contains the raw XML 1.0 name used in the source document, and the \textit{attrs} parameter holds an instance of the \texttt{AttributesNS} interface (see \texttt{The AttributesNS Interface}) containing the attributes of the element. If no namespace is associated with the element, the \textit{uri} component of \textit{name} will be \texttt{None}. The object passed as \textit{attrs} may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the \texttt{copy()} method of the \textit{attrs} object.

Parsers may set the \textit{qname} parameter to \texttt{None}, unless the
feature_namespace_prefixes feature is activated.

ContentHandler.endElementNS(name, qname)
Signals the end of an element in namespace mode.

The name parameter contains the name of the element type, just as with the startElementNS() method, likewise the qname parameter.

ContentHandler.characters(content)
Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information.

content may be a Unicode string or a byte string; the expat reader module produces always Unicode strings.

Note: The earlier SAX 1 interface provided by the Python XML Special Interest Group used a more Java-like interface for this method. Since most parsers used from Python did not take advantage of the older interface, the simpler signature was chosen to replace it. To convert old code to the new interface, use content instead of slicing content with the old offset and length parameters.

ContentHandler.ignorableWhitespace(whitespace)
Receive notification of ignorable whitespace in element content.

Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10): non-validating parsers may also use this method if
they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

**ContentHandler.** `processingInstruction(target, data)`
Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

**ContentHandler.** `skippedEntity(name)`
Receive notification of a skipped entity.

The Parser will invoke this method once for each entity skipped. Non-validating processors may skip entities if they have not seen the declarations (because, for example, the entity was declared in an external DTD subset). All processors may skip external entities, depending on the values of the `feature_external_ges` and the `feature_external_pes` properties.
19.12.2. DTDHandler Objects

**DTDHandler** instances provide the following methods:

```java
DTDHandler notationDecl(name, publicId, systemId)
Handle a notation declaration event.

DTDHandler unparsedEntityDecl(name, publicId, systemId, ndata)
Handle an unparsed entity declaration event.
```
19.12.3. EntityResolver Objects

EntityResolver.\texttt{resolveEntity(publicId, systemId)}

Resolve the system identifier of an entity and return either the system identifier to read from as a string, or an InputSource to read from. The default implementation returns \texttt{systemId}. 
19.12.4. ErrorHandler Objects

Objects with this interface are used to receive error and warning information from the XMLReader. If you create an object that implements this interface, then register the object with your XMLReader, the parser will call the methods in your object to report all warnings and errors. There are three levels of errors available: warnings, (possibly) recoverable errors, and unrecoverable errors. All methods take a SAXParseException as the only parameter. Errors and warnings may be converted to an exception by raising the passed-in exception object.

ErrorHandler.error(exception)
   Called when the parser encounters a recoverable error. If this method does not raise an exception, parsing may continue, but further document information should not be expected by the application. Allowing the parser to continue may allow additional errors to be discovered in the input document.

ErrorHandler.fatalError(exception)
   Called when the parser encounters an error it cannot recover from; parsing is expected to terminate when this method returns.

ErrorHandler.warning(exception)
   Called when the parser presents minor warning information to the application. Parsing is expected to continue when this method returns, and document information will continue to be passed to the application. Raising an exception in this method will cause parsing to end.
New in version 2.0.

The module `xml.sax.saxutils` contains a number of classes and functions that are commonly useful when creating SAX applications, either in direct use, or as base classes.

`xml.sax.saxutils.escape(data[, entities])`

Escape `&`, `<`, and `>` in a string of data.

You can escape other strings of data by passing a dictionary as the optional `entities` parameter. The keys and values must all be strings; each key will be replaced with its corresponding value. The characters `&`, `<` and `>` are always escaped, even if `entities` is provided.

`xml.sax.saxutils.unescape(data[, entities])`

Unescape `&amp;`, `&lt;`, and `&gt;` in a string of data.

You can unescape other strings of data by passing a dictionary as the optional `entities` parameter. The keys and values must all be strings; each key will be replaced with its corresponding value. `&amp;`, `&lt;`, and `&gt;` are always unescaped, even if `entities` is provided.

New in version 2.3.

`xml.sax.saxutils.quoteattr(data[, entities])`

Similar to `escape()`, but also prepares `data` to be used as an attribute value. The return value is a quoted version of `data` with

[19.13. xml.sax.saxutils — SAX Utilities](#)

*New in version 2.0.*

The module `xml.sax.saxutils` contains a number of classes and functions that are commonly useful when creating SAX applications, either in direct use, or as base classes.

`xml.sax.saxutils.escape(data[, entities])`

Escape `&`, `<`, and `>` in a string of data.

You can escape other strings of data by passing a dictionary as the optional `entities` parameter. The keys and values must all be strings; each key will be replaced with its corresponding value. The characters `&`, `<` and `>` are always escaped, even if `entities` is provided.

`xml.sax.saxutils.unescape(data[, entities])`

Unescape `&amp;`, `&lt;`, and `&gt;` in a string of data.

You can unescape other strings of data by passing a dictionary as the optional `entities` parameter. The keys and values must all be strings; each key will be replaced with its corresponding value. `&amp;`, `&lt;`, and `&gt;` are always unescaped, even if `entities` is provided.

*New in version 2.3.*

`xml.sax.saxutils.quoteattr(data[, entities])`

Similar to `escape()`, but also prepares `data` to be used as an attribute value. The return value is a quoted version of `data` with
any additional required replacements. `quoteattr()` will select a quote character based on the content of `data`, attempting to avoid encoding any quote characters in the string. If both single- and double-quote characters are already in `data`, the double-quote characters will be encoded and `data` will be wrapped in double-quotes. The resulting string can be used directly as an attribute value:

```python
>>> print "<element attr=%s>" % quoteattr("ab ' cd " ef")
<element attr="ab ' cd &quot; ef">
```

This function is useful when generating attribute values for HTML or any SGML using the reference concrete syntax.

*New in version 2.2.*

**`class xml.sax.saxutils.XMLGenerator(out[, encoding])`**

This class implements the `ContentHandler` interface by writing SAX events back into an XML document. In other words, using an `XMLGenerator` as the content handler will reproduce the original document being parsed. `out` should be a file-like object which will default to `sys.stdout`. `encoding` is the encoding of the output stream which defaults to `'iso-8859-1'`.

**`class xml.sax.saxutils.XMLFilterBase(base)`**

This class is designed to sit between an `XMLReader` and the client application’s event handlers. By default, it does nothing but pass requests up to the reader and events on to the handlers unmodified, but subclasses can override specific methods to modify the event stream or the configuration requests as they pass through.

```python
xml.sax.saxutils.prepare_input_source(source[, base])
```

This function takes an input source and an optional base URL
and returns a fully resolved `InputSource` object ready for reading. The input source can be given as a string, a file-like object, or an `InputSource` object; parsers will use this function to implement the polymorphic `source` argument to their `parse()` method.
19.14. `xml.sax.xmlreader` — Interface for XML parsers

*New in version 2.0.*

SAX parsers implement the `XMLReader` interface. They are implemented in a Python module, which must provide a function `create_parser()`. This function is invoked by `xml.sax.make_parser()` with no arguments to create a new parser object.

```python
class xml.sax.xmlreader.XMLReader
    Base class which can be inherited by SAX parsers.
```

```python
class xml.sax.xmlreader.IncrementalParser
    In some cases, it is desirable not to parse an input source at once, but to feed chunks of the document as they get available. Note that the reader will normally not read the entire file, but read it in chunks as well; still `parse()` won’t return until the entire document is processed. So these interfaces should be used if the blocking behaviour of `parse()` is not desirable.
```

When the parser is instantiated it is ready to begin accepting data from the feed method immediately. After parsing has been finished with a call to close the reset method must be called to make the parser ready to accept new data, either from feed or using the parse method.

Note that these methods must *not* be called during parsing, that is, after `parse` has been called and before it returns.

By default, the class also implements the parse method of the XMLReader interface using the feed, close and reset methods of the IncrementalParser interface as a convenience to SAX 2.0
driver writers.

**class xml.sax.xmlreader.Locator**

Interface for associating a SAX event with a document location. A locator object will return valid results only during calls to DocumentHandler methods; at any other time, the results are unpredictable. If information is not available, methods may return None.

**class xml.sax.xmlreader.InputSource([systemId])**

Encapsulation of the information needed by the XMLReader to read entities.

This class may include information about the public identifier, system identifier, byte stream (possibly with character encoding information) and/or the character stream of an entity.

Applications will create objects of this class for use in the XMLReader.parse() method and for returning from EntityResolver.resolveEntity.

An InputSource belongs to the application, the XMLReader is not allowed to modify InputSource objects passed to it from the application, although it may make copies and modify those.

**class xml.sax.xmlreader.AttributesImpl(attrs)**

This is an implementation of the Attributes interface (see section The Attributes Interface). This is a dictionary-like object which represents the element attributes in a startElement() call. In addition to the most useful dictionary operations, it supports a number of other methods as described by the interface. Objects of this class should be instantiated by readers; attrs must be a dictionary-like object containing a mapping from attribute names to attribute values.
class xml.sax.xmlreader.AttributesNSImpl(attrs, qnames)

Namespace-aware variant of AttributesImpl, which will be passed to startElementNS(). It is derived from AttributesImpl, but understands attribute names as two-tuples of namespaceURI and localname. In addition, it provides a number of methods expecting qualified names as they appear in the original document. This class implements the AttributesNS interface (see section The AttributesNS Interface).
19.14.1. XMLReader Objects

The **XMLReader** interface supports the following methods:

**XMLReader.**`parse(source)`
- Process an input source, producing SAX events. The `source` object can be a system identifier (a string identifying the input source – typically a file name or a URL), a file-like object, or an `InputSource` object. When `parse()` returns, the input is completely processed, and the parser object can be discarded or reset. As a limitation, the current implementation only accepts byte streams; processing of character streams is for further study.

**XMLReader.**`getContentHandler()`
- Return the current `ContentHandler`.

**XMLReader.**`setContentHandler(handler)`
- Set the current `ContentHandler`. If no `ContentHandler` is set, content events will be discarded.

**XMLReader.**`getDTDHandler()`
- Return the current `DTDHandler`.

**XMLReader.**`setDTDHandler(handler)`
- Set the current `DTDHandler`. If no `DTDHandler` is set, DTD events will be discarded.

**XMLReader.**`getEntityResolver()`
- Return the current `EntityResolver`.

**XMLReader.**`setEntityResolver(handler)`
- Set the current `EntityResolver`. If no `EntityResolver` is set, attempts to resolve an external entity will result in opening the
system identifier for the entity, and fail if it is not available.

**XMLReader**. `getErrorHandler()`

Return the current `ErrorHandler`.

**XMLReader**. `setErrorHandler(handler)`

Set the current error handler. If no `ErrorHandler` is set, errors will be raised as exceptions, and warnings will be printed.

**XMLReader**. `setLocale(locale)`

Allow an application to set the locale for errors and warnings.

SAX parsers are not required to provide localization for errors and warnings; if they cannot support the requested locale, however, they must raise a SAX exception. Applications may request a locale change in the middle of a parse.

**XMLReader**. `getFeature(featurename)`

Return the current setting for feature `featurename`. If the feature is not recognized, `SAXNotRecognizedException` is raised. The well-known featurenames are listed in the module `xml.sax.handler`.

**XMLReader**. `setFeature(featurename, value)`

Set the `featurename` to `value`. If the feature is not recognized, `SAXNotRecognizedException` is raised. If the feature or its setting is not supported by the parser, `SAXNotSupportedException` is raised.

**XMLReader**. `getProperty(propertyname)`

Return the current setting for property `propertyname`. If the property is not recognized, a `SAXNotRecognizedException` is raised. The well-known propertynames are listed in the module `xml.sax.handler`.

**XMLReader**. `setProperty(propertyname, value)`
Set the *propertyname* to *value*. If the property is not recognized, `SAXNotRecognizedException` is raised. If the property or its setting is not supported by the parser, `SAXNotSupportedException` is raised.
19.14.2. IncrementalParser Objects

Instances of `IncrementalParser` offer the following additional methods:

IncrementalParser. `feed(data)`
Process a chunk of `data`.

IncrementalParser. `close()`
Assume the end of the document. That will check well-formedness conditions that can be checked only at the end, invoke handlers, and may clean up resources allocated during parsing.

IncrementalParser. `reset()`
This method is called after close has been called to reset the parser so that it is ready to parse new documents. The results of calling parse or feed after close without calling reset are undefined.
Instances of `Locator` provide these methods:

`Locator.getColumnNumber()`  
Return the column number where the current event begins.

`Locator.getLineNumber()`  
Return the line number where the current event begins.

`Locator.getPublicId()`  
Return the public identifier for the current event.

`Locator.getSystemId()`  
Return the system identifier for the current event.
19.14.4. InputSource Objects

InputSource. **setPublicId**(id)
Sets the public identifier of this InputSource.

InputSource. **getPublicId**()
Returns the public identifier of this InputSource.

InputSource. **setSystemId**(id)
Sets the system identifier of this InputSource.

InputSource. **getSystemId**()
Returns the system identifier of this InputSource.

InputSource. **setEncoding**(encoding)
Sets the character encoding of this InputSource.

The encoding must be a string acceptable for an XML encoding declaration (see section 4.3.3 of the XML recommendation).

The encoding attribute of the InputSource is ignored if the InputSource also contains a character stream.

InputSource. **getEncoding**()
Get the character encoding of this InputSource.

InputSource. **setByteStream**(bytefile)
Set the byte stream (a Python file-like object which does not perform byte-to-character conversion) for this input source.

The SAX parser will ignore this if there is also a character stream specified, but it will use a byte stream in preference to opening a URI connection itself.
If the application knows the character encoding of the byte stream, it should set it with the setEncoding method.

**InputSource**. **getByteStream()**
Get the byte stream for this input source.

The getEncoding method will return the character encoding for this byte stream, or `None` if unknown.

**InputSource**. **setCharacterStream(charfile)**
Set the character stream for this input source. (The stream must be a Python 1.6 Unicode-wrapped file-like that performs conversion to Unicode strings.)

If there is a character stream specified, the SAX parser will ignore any byte stream and will not attempt to open a URI connection to the system identifier.

**InputSource**. **getCharacterStream()**
Get the character stream for this input source.
19.14.5. The Attributes Interface

Attributes objects implement a portion of the mapping protocol, including the methods copy(), get(), has_key(), items(), keys(), and values(). The following methods are also provided:

**Attributes.getLength()**
Return the number of attributes.

**Attributes(getNames())**
Return the names of the attributes.

**Attributes.getType(name)**
Returns the type of the attribute name, which is normally 'CDATA'.

**Attributes.getValue(name)**
Return the value of attribute name.
19.14.6. The **AttributesNS** Interface

This interface is a subtype of the **Attributes** interface (see section The Attributes Interface). All methods supported by that interface are also available on **AttributesNS** objects.

The following methods are also available:

- **AttributesNS.** `getValueByQName(name)`
  
  Return the value for a qualified name.

- **AttributesNS.** `getNameByQName(name)`
  
  Return the (namespace, localname) pair for a qualified name.

- **AttributesNS.** `getQNameByName(name)`
  
  Return the qualified name for a (namespace, localname) pair.

- **AttributesNS.** `getQNames()`
  
  Return the qualified names of all attributes.
Warning: The `pyexpat` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

New in version 2.0.

The `xml.parsers.expat` module is a Python interface to the Expat non-validating XML parser. The module provides a single extension type, `xmlparser`, that represents the current state of an XML parser. After an `xmlparser` object has been created, various attributes of the object can be set to handler functions. When an XML document is then fed to the parser, the handler functions are called for the character data and markup in the XML document.

This module uses the `pyexpat` module to provide access to the Expat parser. Direct use of the `pyexpat` module is deprecated.

This module provides one exception and one type object:

> exception `xml.parsers.expat.ExpatError`
> The exception raised when Expat reports an error. See section `ExpatError Exceptions` for more information on interpreting Expat errors.

> exception `xml.parsers.expat.error`
> Alias for `ExpatError`.

`xml.parsers.expat.XMLParserType`
> The type of the return values from the `ParserCreate()` function.
The `xml.parsers.expat` module contains two functions:

`xml.parsers.expat.ErrorString(errno)`

Returns an explanatory string for a given error number `errno`.

`xml.parsers.expat.ParserCreate([encoding[, namespace_separator]])`  
Creates and returns a new `xmlparser` object. `encoding`, if specified, must be a string naming the encoding used by the XML data. Expat doesn’t support as many encodings as Python does, and its repertoire of encodings can’t be extended; it supports UTF-8, UTF-16, ISO-8859-1 (Latin1), and ASCII. If `encoding` [1] is given it will override the implicit or explicit encoding of the document.

Expat can optionally do XML namespace processing for you, enabled by providing a value for `namespace_separator`. The value must be a one-character string; a `ValueError` will be raised if the string has an illegal length (None is considered the same as omission). When namespace processing is enabled, element type names and attribute names that belong to a namespace will be expanded. The element name passed to the element handlers `StartElementHandler` and `EndElementHandler` will be the concatenation of the namespace URI, the namespace separator character, and the local part of the name. If the namespace separator is a zero byte (`chr(0)`) then the namespace URI and the local part will be concatenated without any separator.

For example, if `namespace_separator` is set to a space character (' ') and the following document is parsed:

```xml
<?xml version="1.0"?>
<root xmlns = "http://default-namespace.org/"
     xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
```
StartElementHandler will receive the following strings for each element:

- http://default-namespace.org/ root
- http://www.python.org/ns/ elem1
- elem2

Due to limitations in the Expat library used by pyexpat, the xmlparser instance returned can only be used to parse a single XML document. Call ParserCreate for each document to provide unique parser instances.

See also:

The Expat XML Parser
Home page of the Expat project.
19.15.1. XMLParser Objects

xmlparser objects have the following methods:

xmlparser.\texttt{Parse}(data[, isfinal])

Parses the contents of the string \textit{data}, calling the appropriate handler functions to process the parsed data. \textit{isfinal} must be true on the final call to this method; it allows the parsing of a single file in fragments, not the submission of multiple files. \textit{data} can be the empty string at any time.

xmlparser.\texttt{ParseFile}(file)

Parse XML data reading from the object \textit{file}. \textit{file} only needs to provide the \texttt{read(nbytes)} method, returning the empty string when there's no more data.

xmlparser.\texttt{SetBase}(base)

Sets the base to be used for resolving relative URIs in system identifiers in declarations. Resolving relative identifiers is left to the application: this value will be passed through as the \textit{base} argument to the \texttt{ExternalEntityRefHandler()}, \texttt{NotationDeclHandler()}, and \texttt{UnparsedEntityDeclHandler()} functions.

xmlparser.\texttt{GetBase}()

Returns a string containing the base set by a previous call to \texttt{SetBase()}, or \texttt{None} if \texttt{SetBase()} hasn't been called.

xmlparser.\texttt{GetInputContext}()

Returns the input data that generated the current event as a string. The data is in the encoding of the entity which contains the text. When called while an event handler is not active, the return value is \texttt{None}.
New in version 2.1.

`xmlparser.ExternalEntityParserCreate(context[, encoding])`
Create a “child” parser which can be used to parse an external parsed entity referred to by content parsed by the parent parser. The context parameter should be the string passed to the ExternalEntityRefHandler() handler function, described below. The child parser is created with the ordered_attributes, returns_unicode and specified_attributes set to the values of this parser.

`xmlparser.SetParamEntityParsing(flag)`
Control parsing of parameter entities (including the external DTD subset). Possible flag values are XML_PARAM_ENTITY_PARSING_NEVER, XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE and XML_PARAM_ENTITY_PARSING_ALWAYS. Return true if setting the flag was successful.

`xmlparser.UseForeignDTD([flag])`
Calling this with a true value for flag (the default) will cause Expat to call the ExternalEntityRefHandler with None for all arguments to allow an alternate DTD to be loaded. If the document does not contain a document type declaration, the ExternalEntityRefHandler will still be called, but the StartDoctypeDeclHandler and EndDoctypeDeclHandler will not be called.

Passing a false value for flag will cancel a previous call that passed a true value, but otherwise has no effect.

This method can only be called before the Parse() or ParseFile() methods are called; calling it after either of those
have been called causes `ExpatError` to be raised with the code set to `errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING`.

*New in version 2.3.*

`xmlparser` objects have the following attributes:

`xmlparser.buffer_size`

The size of the buffer used when `buffer_text` is true. A new buffer size can be set by assigning a new integer value to this attribute. When the size is changed, the buffer will be flushed.

*New in version 2.3.*

*Changed in version 2.6:* The buffer size can now be changed.

`xmlparser.buffer_text`

Setting this to true causes the `xmlparser` object to buffer textual content returned by Expat to avoid multiple calls to the `CharacterDataHandler()` callback whenever possible. This can improve performance substantially since Expat normally breaks character data into chunks at every line ending. This attribute is false by default, and may be changed at any time.

*New in version 2.3.*

`xmlparser.buffer_used`

If `buffer_text` is enabled, the number of bytes stored in the buffer. These bytes represent UTF-8 encoded text. This attribute has no meaningful interpretation when `buffer_text` is false.

*New in version 2.3.*

`xmlparser.ordered_attributes`

Setting this attribute to a non-zero integer causes the attributes to
be reported as a list rather than a dictionary. The attributes are presented in the order found in the document text. For each attribute, two list entries are presented: the attribute name and the attribute value. (Older versions of this module also used this format.) By default, this attribute is false; it may be changed at any time.

*New in version 2.1.*

**xmlparser**. `returns_unicode`

If this attribute is set to a non-zero integer, the handler functions will be passed Unicode strings. If `returns_unicode` is `False`, 8-bit strings containing UTF-8 encoded data will be passed to the handlers. This is `True` by default when Python is built with Unicode support.

*Changed in version 1.6:* Can be changed at any time to affect the result type.

**xmlparser**. `specified_attributes`

If set to a non-zero integer, the parser will report only those attributes which were specified in the document instance and not those which were derived from attribute declarations. Applications which set this need to be especially careful to use what additional information is available from the declarations as needed to comply with the standards for the behavior of XML processors. By default, this attribute is false; it may be changed at any time.

*New in version 2.1.*

The following attributes contain values relating to the most recent error encountered by an `xmlparser` object, and will only have correct values once a call to `Parse()` or `ParseFile()` has raised an `xml.parsers.expat.ExpatError` exception.
xmlparser. **ErrorByteIndex**

Byte index at which an error occurred.

xmlparser. **ErrorCode**

Numeric code specifying the problem. This value can be passed to the `ErrorString()` function, or compared to one of the constants defined in the `errors` object.

xmlparser. **ErrorColumnNumber**

Column number at which an error occurred.

xmlparser. **ErrorLineNumber**

Line number at which an error occurred.

The following attributes contain values relating to the current parse location in an `xmlparser` object. During a callback reporting a parse event they indicate the location of the first of the sequence of characters that generated the event. When called outside of a callback, the position indicated will be just past the last parse event (regardless of whether there was an associated callback).

*New in version 2.4.*

xmlparser. **CurrentByteIndex**

Current byte index in the parser input.

xmlparser. **CurrentColumnNumber**

Current column number in the parser input.

xmlparser. **CurrentLineNumber**

Current line number in the parser input.

Here is the list of handlers that can be set. To set a handler on an `xmlparser` object `o`, use `o.handlername = func`. `handlername` must be taken from the following list, and `func` must be a callable object accepting the correct number of arguments. The arguments are all
strings, unless otherwise stated.

xmlparser.**XmlDeclHandler**(version, encoding, standalone)

Called when the XML declaration is parsed. The XML declaration is the (optional) declaration of the applicable version of the XML recommendation, the encoding of the document text, and an optional “standalone” declaration. `version` and `encoding` will be strings of the type dictated by the `returns_unicode` attribute, and `standalone` will be 1 if the document is declared standalone, 0 if it is declared not to be standalone, or -1 if the standalone clause was omitted. This is only available with Expat version 1.95.0 or newer.

*New in version 2.1.*

xmlparser.**StartDoctypeDeclHandler**(doctypeName, systemId, publicId, has_internal_subset)

Called when Expat begins parsing the document type declaration (`<!DOCTYPE ...`). The `doctypeName` is provided exactly as presented. The `systemId` and `publicId` parameters give the system and public identifiers if specified, or `None` if omitted. `has_internal_subset` will be true if the document contains and internal document declaration subset. This requires Expat version 1.2 or newer.

xmlparser.**EndDoctypeDeclHandler**()

Called when Expat is done parsing the document type declaration. This requires Expat version 1.2 or newer.

xmlparser.**ElementDeclHandler**(name, model)

Called once for each element type declaration. `name` is the name of the element type, and `model` is a representation of the content model.

xmlparser.**AttlistDeclHandler**(elname, attname, type, default,
required)

Called for each declared attribute for an element type. If an attribute list declaration declares three attributes, this handler is called three times, once for each attribute. elname is the name of the element to which the declaration applies and attname is the name of the attribute declared. The attribute type is a string passed as type; the possible values are 'CDATA', 'ID', 'IDREF', ...

... default gives the default value for the attribute used when the attribute is not specified by the document instance, or None if there is no default value (#IMPLIED values). If the attribute is required to be given in the document instance, required will be true. This requires Expat version 1.95.0 or newer.

xmlparser. **StartElementHandler**(name, attributes)

Called for the start of every element. name is a string containing the element name, and attributes is a dictionary mapping attribute names to their values.

xmlparser. **EndElementHandler**(name)

Called for the end of every element.

xmlparser. **ProcessingInstructionHandler**(target, data)

Called for every processing instruction.

xmlparser. **CharacterDataHandler**(data)

Called for character data. This will be called for normal character data, CDATA marked content, and ignorable whitespace. Applications which must distinguish these cases can use the StartCdataSectionHandler, EndCdataSectionHandler, and ElementDeclHandler callbacks to collect the required information.

xmlparser. **UnparsedEntityDeclHandler**(entityName, base, systemId, publicId, notationName)

Called for unparsed (NDATA) entity declarations. This is only present for version 1.2 of the Expat library; for more recent
versions, use `EntityDeclHandler` instead. (The underlying function in the Expat library has been declared obsolete.)

```python
xmlparser.EntityDeclHandler(entityName, is_parameter_entity, value, base, systemId, publicId, notationName)
```

Called for all entity declarations. For parameter and internal entities, `value` will be a string giving the declared contents of the entity; this will be `None` for external entities. The `notationName` parameter will be `None` for parsed entities, and the name of the notation for unparsed entities. `is_parameter_entity` will be true if the entity is a parameter entity or false for general entities (most applications only need to be concerned with general entities). This is only available starting with version 1.95.0 of the Expat library.

*New in version 2.1.*

```python
xmlparser.NotationDeclHandler(notationName, base, systemId, publicId)
```

Called for notation declarations. `notationName`, `base`, and `systemId`, and `publicId` are strings if given. If the public identifier is omitted, `publicId` will be `None`.

```python
xmlparser.StartNamespaceDeclHandler(prefix, uri)
```

Called when an element contains a namespace declaration. Namespace declarations are processed before the `StartElementHandler` is called for the element on which declarations are placed.

```python
xmlparser.EndNamespaceDeclHandler(prefix)
```

Called when the closing tag is reached for an element that contained a namespace declaration. This is called once for each namespace declaration on the element in the reverse of the order for which the `StartNamespaceDeclHandler` was called to indicate the start of each namespace declaration’s scope. Calls to this
handler are made after the corresponding `EndElementHandler` for the end of the element.

`xmlparser.CommentHandler(data)`
Called for comments. `data` is the text of the comment, excluding the leading `'<!-'` and trailing `'->'`.

`xmlparser.StartCdataSectionHandler()`
Called at the start of a CDATA section. This and `EndCdataSectionHandler` are needed to be able to identify the syntactical start and end for CDATA sections.

`xmlparser.EndCdataSectionHandler()`
Called at the end of a CDATA section.

`xmlparser.DefaultHandler(data)`
Called for any characters in the XML document for which no applicable handler has been specified. This means characters that are part of a construct which could be reported, but for which no handler has been supplied.

`xmlparser.DefaultHandlerExpand(data)`
This is the same as the `DefaultHandler()`, but doesn’t inhibit expansion of internal entities. The entity reference will not be passed to the default handler.

`xmlparser.NotStandaloneHandler()`
Called if the XML document hasn’t been declared as being a standalone document. This happens when there is an external subset or a reference to a parameter entity, but the XML declaration does not set standalone to `yes` in an XML declaration. If this handler returns `0`, then the parser will raise an `XML_ERROR_NOT_STANDALONE` error. If this handler is not set, no exception is raised by the parser for this condition.
xmlparser. `ExternalEntityRefHandler(context, base, systemId, publicId)`

Called for references to external entities. `base` is the current base, as set by a previous call to `SetBase()`. The public and system identifiers, `systemId` and `publicId`, are strings if given; if the public identifier is not given, `publicId` will be `None`. The `context` value is opaque and should only be used as described below.

For external entities to be parsed, this handler must be implemented. It is responsible for creating the sub-parser using `ExternalEntityParserCreate(context)`, initializing it with the appropriate callbacks, and parsing the entity. This handler should return an integer; if it returns `0`, the parser will raise an `XML_ERROR_EXTERNAL_ENTITY_HANDLING` error, otherwise parsing will continue.

If this handler is not provided, external entities are reported by the `DefaultHandler` callback, if provided.
19.15.2. ExpatError Exceptions

ExpatError exceptions have a number of interesting attributes:

ExpatError.code
Expat’s internal error number for the specific error. This will match one of the constants defined in the errors object from this module.

*New in version 2.1.*

ExpatError.lineno
Line number on which the error was detected. The first line is numbered 1.

*New in version 2.1.*

ExpatError.offset
Character offset into the line where the error occurred. The first column is numbered 0.

*New in version 2.1.*
The following program defines three handlers that just print out their arguments.

```python
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print 'Start element:', name, attrs

def end_element(name):
    print 'End element:', name

def char_data(data):
    print 'Character data:', repr(data)

p = xml.parsers.expat.ParserCreate()
p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

The output from this program is:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```
19.15.4. Content Model Descriptions

Content models are described using nested tuples. Each tuple contains four values: the type, the quantifier, the name, and a tuple of children. Children are simply additional content model descriptions.

The values of the first two fields are constants defined in the `model` object of the `xml.parsers.expat` module. These constants can be collected in two groups: the model type group and the quantifier group.

The constants in the model type group are:

```python
xml.parsers.expat.XML_CTYPE_ANY

The element named by the model name was declared to have a content model of ANY.
```

```python
xml.parsers.expat.XML_CTYPE_CHOICE

The named element allows a choice from a number of options; this is used for content models such as `(A | B | C)`.
```

```python
xml.parsers.expat.XML_CTYPE_EMPTY

Elements which are declared to be EMPTY have this model type.
```

```python
xml.parsers.expat.XML_CTYPE_MIXED
xml.parsers.expat.XML_CTYPE_NAME
xml.parsers.expat.XML_CTYPE_SEQ

Models which represent a series of models which follow one after the other are indicated with this model type. This is used for models such as `(A, B, C)`.
```

The constants in the quantifier group are:
xml.parsers.expat.XML_CQUANT_NONE
   No modifier is given, so it can appear exactly once, as for \textit{A}.

xml.parsers.expat.XML_CQUANT_OPT
   The model is optional: it can appear once or not at all, as for \textit{A?}.

xml.parsers.expat.XML_CQUANT_PLUS
   The model must occur one or more times (like \textit{A+}).

xml.parsers.expat.XML_CQUANT_REP
   The model must occur zero or more times, as for \textit{A*}.
The following constants are provided in the `errors` object of the `xml.parsers.expat` module. These constants are useful in interpreting some of the attributes of the `ExpatError` exception objects raised when an error has occurred.

The `errors` object has the following attributes:

- `xml.parsers.expat.XML_ERROR_ASYNC_ENTITY`
- `xml.parsers.expat.XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF`
  - An entity reference in an attribute value referred to an external entity instead of an internal entity.
- `xml.parsers.expat.XML_ERROR_BAD_CHAR_REF`
  - A character reference referred to a character which is illegal in XML (for example, character \&, or ‘&\#0;’).
- `xml.parsers.expat.XML_ERROR_BINARY_ENTITY_REF`
  - An entity reference referred to an entity which was declared with a notation, so cannot be parsed.
- `xml.parsers.expat.XML_ERROR_DUPLICATE_ATTRIBUTE`
  - An attribute was used more than once in a start tag.
- `xml.parsers.expat.XML_ERROR_INCORRECT_ENCODING`
- `xml.parsers.expat.XML_ERROR_INVALID_TOKEN`
  - Raised when an input byte could not properly be assigned to a character; for example, a NUL byte (value \&) in a UTF-8 input stream.
- `xml.parsers.expat.XML_ERROR_JUNK_AFTER_DOC_ELEMENT`
  - Something other than whitespace occurred after the document
element.

xml.parsers.expat.XML_ERROR_MISPLACED_XML_PI
An XML declaration was found somewhere other than the start of the input data.

xml.parsers.expat.XML_ERROR_NO_ELEMENTS
The document contains no elements (XML requires all documents to contain exactly one top-level element).

xml.parsers.expat.XML_ERROR_NO_MEMORY
Expat was not able to allocate memory internally.

xml.parsers.expat.XML_ERROR_PARAM_ENTITY_REF
A parameter entity reference was found where it was not allowed.

xml.parsers.expat.XML_ERROR_PARTIAL_CHAR
An incomplete character was found in the input.

xml.parsers.expat.XML_ERROR_RECURSIVE_ENTITY_REF
An entity reference contained another reference to the same entity; possibly via a different name, and possibly indirectly.

xml.parsers.expat.XML_ERROR_SYNTAX
Some unspecified syntax error was encountered.

xml.parsers.expat.XML_ERROR_TAG_MISMATCH
An end tag did not match the innermost open start tag.

xml.parsers.expat.XML_ERROR_UNCLOSED_TOKEN
Some token (such as a start tag) was not closed before the end of the stream or the next token was encountered.

xml.parsers.expat.XML_ERROR_UNDEFINED_ENTITY
A reference was made to an entity which was not defined.

xml.parsers.expat.XML_ERROR_UNKNOWN_ENCODING
The document encoding is not supported by Expat.
xml.parsers.expat.**XML_ERROR_UNCLOSED_CDATA_SECTION**
A CDATA marked section was not closed.

xml.parsers.expat.**XML_ERROR_EXTERNAL_ENTITY_HANDLING**

xml.parsers.expat.**XML_ERROR_NOT_STANDALONE**
The parser determined that the document was not “standalone” though it declared itself to be in the XML declaration, and the `NotStandaloneHandler` was set and returned 0.

xml.parsers.expat.**XML_ERROR_UNEXPECTED_STATE**

xml.parsers.expat.**XML_ERROR_ENTITY_DECLARED_IN_PE**

xml.parsers.expat.**XML_ERROR_FEATURE_REQUIRES_XML_DTD**
An operation was requested that requires DTD support to be compiled in, but Expat was configured without DTD support. This should never be reported by a standard build of the `xml.parsers.expat` module.

xml.parsers.expat.**XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING**
A behavioral change was requested after parsing started that can only be changed before parsing has started. This is (currently) only raised by `UseForeignDTD()`.

xml.parsers.expat.**XML_ERROR_UNBOUND_PREFIX**
An undeclared prefix was found when namespace processing was enabled.

xml.parsers.expat.**XML_ERROR_UNDECLARING_PREFIX**
The document attempted to remove the namespace declaration associated with a prefix.

xml.parsers.expat.**XML_ERROR_INCOMPLETE_PE**
A parameter entity contained incomplete markup.

xml.parsers.expat.**XML_ERROR_XML_DECL**
The document contained no document element at all.

xml.parsers.expat.XML_ERROR_TEXT_DECL

There was an error parsing a text declaration in an external entity.

xml.parsers.expat.XML_ERROR_PUBLICID

Characters were found in the public id that are not allowed.

xml.parsers.expat.XML_ERROR_SUSPENDED

The requested operation was made on a suspended parser, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.XML_ERROR_NOT_SUSPENDED

An attempt to resume the parser was made when the parser had not been suspended.

xml.parsers.expat.XML_ERROR_ABORTED

This should not be reported to Python applications.

xml.parsers.expat.XML_ERROR_FINISHED

The requested operation was made on a parser which was finished parsing input, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.XML_ERROR_SUSPEND_PE

Footnotes

[1] The encoding string included in XML output should conform to the appropriate standards. For example, “UTF-8” is valid, but “UTF8” is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.
20. Internet Protocols and Support

The modules described in this chapter implement Internet protocols and support for related technology. They are all implemented in Python. Most of these modules require the presence of the system-dependent module *socket*, which is currently supported on most popular platforms. Here is an overview:

- **20.1. webbrowser** — Convenient Web-browser controller
  - 20.1.1. Browser Controller Objects
- **20.2. cgi** — Common Gateway Interface support
  - 20.2.1. Introduction
  - 20.2.2. Using the cgi module
  - 20.2.3. Higher Level Interface
  - 20.2.4. Old classes
  - 20.2.5. Functions
  - 20.2.6. Caring about security
  - 20.2.7. Installing your CGI script on a Unix system
  - 20.2.8. Testing your CGI script
  - 20.2.9. Debugging CGI scripts
  - 20.2.10. Common problems and solutions
- **20.3. cgitb** — Traceback manager for CGI scripts
- **20.4. wsgiref** — WSGI Utilities and Reference Implementation
  - 20.4.1. *wsgiref.util* — WSGI environment utilities
  - 20.4.2. *wsgiref.headers* — WSGI response header tools
  - 20.4.3. *wsgiref.simple_server* — a simple WSGI HTTP server
  - 20.4.4. *wsgiref.validate* — WSGI conformance checker
  - 20.4.5. *wsgiref.handlers* — server/gateway base classes
  - 20.4.6. Examples
- **20.5. urllib** — Open arbitrary resources by URL
  - 20.5.1. High-level interface
20.5.2. Utility functions
20.5.3. URL Opener objects
20.5.4. urllib Restrictions
20.5.5. Examples

20.6. urllib2 — extensible library for opening URLs
20.6.1. Request Objects
20.6.2. OpenerDirector Objects
20.6.3. BaseHandler Objects
20.6.4. HTTPRedirectHandler Objects
20.6.5. HTTPCookieProcessor Objects
20.6.6. ProxyHandler Objects
20.6.7. HTTPPasswordMgr Objects
20.6.8. AbstractBasicAuthHandler Objects
20.6.9. HTTPBasicAuthHandler Objects
20.6.10. ProxyBasicAuthHandler Objects
20.6.11. AbstractDigestAuthHandler Objects
20.6.12. HTTPDigestAuthHandler Objects
20.6.13. ProxyDigestAuthHandler Objects
20.6.14. HTTPHandler Objects
20.6.15. HTTPSHandler Objects
20.6.16. FileHandler Objects
20.6.17. FTPHandler Objects
20.6.18. CacheFTPHandler Objects
20.6.19. UnknownHandler Objects
20.6.20. HTTPErrorProcessor Objects
20.6.21. Examples

20.7. httplib — HTTP protocol client
20.7.1. HTTPConnection Objects
20.7.2. HTTPResponse Objects
20.7.3. Examples

20.8. ftplib — FTP protocol client
20.8.1. FTP Objects
20.8.2. FTP_TLS Objects

20.9. poplib — POP3 protocol client
• 20.9.1. POP3 Objects
  • 20.9.2. POP3 Example
• 20.10. imaplib — IMAP4 protocol client
  • 20.10.1. IMAP4 Objects
  • 20.10.2. IMAP4 Example
• 20.11. nntplib — NNTP protocol client
  • 20.11.1. NNTP Objects
• 20.12. smtplib — SMTP protocol client
  • 20.12.1. SMTP Objects
  • 20.12.2. SMTP Example
• 20.13. smtpd — SMTP Server
  • 20.13.1. SMTPServer Objects
  • 20.13.2. DebuggingServer Objects
  • 20.13.3. PureProxy Objects
  • 20.13.4. MailmanProxy Objects
• 20.14. telnetlib — Telnet client
  • 20.14.1. Telnet Objects
  • 20.14.2. Telnet Example
• 20.15. uuid — UUID objects according to RFC 4122
  • 20.15.1. Example
• 20.16. urlparse — Parse URLs into components
  • 20.16.1. Results of urlparse() and urlsplit()
• 20.17. SocketServer — A framework for network servers
  • 20.17.1. Server Creation Notes
  • 20.17.2. Server Objects
  • 20.17.3. Request Handler Objects
  • 20.17.4. Examples
    • 20.17.4.1. SocketServer.TCPServer Example
    • 20.17.4.2. SocketServer.UDPServer Example
    • 20.17.4.3. Asynchronous Mixins
• 20.18. BaseHTTPServer — Basic HTTP server
  • 20.18.1. More examples
• 20.19. SimpleHTTPServer — Simple HTTP request handler
- 20.20. CGIHTTPServer — CGI-capable HTTP request handler
- 20.21. cookielib — Cookie handling for HTTP clients
  - 20.21.2. FileCookieJar subclasses and co-operation with web browsers
  - 20.21.3. CookiePolicy Objects
  - 20.21.4. DefaultCookiePolicy Objects
  - 20.21.5. Cookie Objects
  - 20.21.6. Examples
- 20.22. Cookie — HTTP state management
  - 20.22.1. Cookie Objects
  - 20.22.2. Morsel Objects
  - 20.22.3. Example
- 20.23. xmlrpc_clientlib — XML-RPC client access
  - 20.23.1. ServerProxy Objects
  - 20.23.2. Boolean Objects
  - 20.23.3. DateTime Objects
  - 20.23.4. Binary Objects
  - 20.23.5. Fault Objects
  - 20.23.6. ProtocolError Objects
  - 20.23.7. MultiCall Objects
  - 20.23.8. Convenience Functions
  - 20.23.9. Example of Client Usage
  - 20.23.10. Example of Client and Server Usage
- 20.24. SimpleXMLRPCServer — Basic XML-RPC server
  - 20.24.1. SimpleXMLRPCServer Objects
    - 20.24.1.1. SimpleXMLRPCServer Example
  - 20.24.2. CGIXMLRPCRequestHandler
- 20.25. DocXMLRPCServer — Self-documenting XML-RPC server
  - 20.25.1. DocXMLRPCServer Objects
  - 20.25.2. DocCGIXMLRPCRequestHandler
The `webbrowser` module provides a high-level interface to allow displaying Web-based documents to users. Under most circumstances, simply calling the `open()` function from this module will do the right thing.

Under Unix, graphical browsers are preferred under X11, but text-mode browsers will be used if graphical browsers are not available or an X11 display isn't available. If text-mode browsers are used, the calling process will block until the user exits the browser.

If the environment variable `BROWSER` exists, it is interpreted to override the platform default list of browsers, as an `os.pathsep`-separated list of browsers to try in order. When the value of a list part contains the string `%s`, then it is interpreted as a literal browser command line to be used with the argument URL substituted for `%s`; if the part does not contain `%s`, it is simply interpreted as the name of the browser to launch. [1]

For non-Unix platforms, or when a remote browser is available on Unix, the controlling process will not wait for the user to finish with the browser, but allow the remote browser to maintain its own windows on the display. If remote browsers are not available on Unix, the controlling process will launch a new browser and wait.

The script `webbrowser` can be used as a command-line interface for the module. It accepts a URL as the argument. It accepts the following optional parameters: `-n` opens the URL in a new browser.
window, if possible; -t opens the URL in a new browser page ("tab"). The options are, naturally, mutually exclusive. Usage example:

```python
python -m webbrowser -t "http://www.python.org"
```

The following exception is defined:

```
exception webbrowser. Error
```

Exception raised when a browser control error occurs.

The following functions are defined:

```
webbrowser.open(url, new=0, autoraise=True)
```

Display url using the default browser. If new is 0, the url is opened in the same browser window if possible. If new is 1, a new browser window is opened if possible. If new is 2, a new browser page ("tab") is opened if possible. If autoraise is True, the window is raised if possible (note that under many window managers this will occur regardless of the setting of this variable).

Note that on some platforms, trying to open a filename using this function, may work and start the operating system’s associated program. However, this is neither supported nor portable.

*Changed in version 2.5:* new can now be 2.

```
webbrowser.open_new(url)
```

Open url in a new window of the default browser, if possible, otherwise, open url in the only browser window.

```
webbrowser.open_new_tab(url)
```

Open url in a new page ("tab") of the default browser, if possible, otherwise equivalent to `open_new()`.

*New in version 2.5.*
webbrowser.get([name])

Return a controller object for the browser type name. If name is empty, return a controller for a default browser appropriate to the caller’s environment.

webbrowser.register(name, constructor[, instance])

Register the browser type name. Once a browser type is registered, the get() function can return a controller for that browser type. If instance is not provided, or is None, constructor will be called without parameters to create an instance when needed. If instance is provided, constructor will never be called, and may be None.

This entry point is only useful if you plan to either set the BROWSER variable or call get() with a nonempty argument matching the name of a handler you declare.

A number of browser types are predefined. This table gives the type names that may be passed to the get() function and the corresponding instantiations for the controller classes, all defined in this module.

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Class Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>'mozilla'</td>
<td>Mozilla('mozilla')</td>
<td></td>
</tr>
<tr>
<td>'firefox'</td>
<td>Mozilla('mozilla')</td>
<td></td>
</tr>
<tr>
<td>'netscape'</td>
<td>Mozilla('netscape')</td>
<td></td>
</tr>
<tr>
<td>'galeon'</td>
<td>Galeon('galeon')</td>
<td></td>
</tr>
<tr>
<td>'epiphany'</td>
<td>Galeon('epiphany')</td>
<td></td>
</tr>
<tr>
<td>'skipstone'</td>
<td>BackgroundBrowser('skipstone')</td>
<td></td>
</tr>
<tr>
<td>'kfmclient'</td>
<td>Konqueror()</td>
<td>(1)</td>
</tr>
<tr>
<td>'konqueror'</td>
<td>Konqueror()</td>
<td>(1)</td>
</tr>
<tr>
<td>'kfm'</td>
<td>Konqueror()</td>
<td>(1)</td>
</tr>
<tr>
<td>'mosaic'</td>
<td>BackgroundBrowser('mosaic')</td>
<td></td>
</tr>
<tr>
<td>Browser</td>
<td>Function</td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------------------------------</td>
<td></td>
</tr>
<tr>
<td>'opera'</td>
<td>Opera()</td>
<td></td>
</tr>
<tr>
<td>'grail'</td>
<td>Grail()</td>
<td></td>
</tr>
<tr>
<td>'links'</td>
<td>GenericBrowser('links')</td>
<td></td>
</tr>
<tr>
<td>'elinks'</td>
<td>Elinks('elinks')</td>
<td></td>
</tr>
<tr>
<td>'lynx'</td>
<td>GenericBrowser('lynx')</td>
<td></td>
</tr>
<tr>
<td>'w3m'</td>
<td>GenericBrowser('w3m')</td>
<td></td>
</tr>
<tr>
<td>'windows-default'</td>
<td>WindowsDefault</td>
<td></td>
</tr>
<tr>
<td>'macosx'</td>
<td>MacOSX('default')</td>
<td></td>
</tr>
<tr>
<td>'safari'</td>
<td>MacOSX('safari')</td>
<td></td>
</tr>
<tr>
<td>'google-chrome'</td>
<td>Chrome('google-chrome')</td>
<td></td>
</tr>
<tr>
<td>'chrome'</td>
<td>Chrome('chrome')</td>
<td></td>
</tr>
<tr>
<td>'chromium'</td>
<td>Chromium('chromium')</td>
<td></td>
</tr>
<tr>
<td>'chromium-browser'</td>
<td>Chromium('chromium-browser')</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. “Konqueror” is the file manager for the KDE desktop environment for Unix, and only makes sense to use if KDE is running. Some way of reliably detecting KDE would be nice; the `KDEDIR` variable is not sufficient. Note also that the name “kfm” is used even when using the `konqueror` command with KDE 2 — the implementation selects the best strategy for running Konqueror.

2. Only on Windows platforms.
3. Only on Mac OS X platform.
4. Support for Chrome/Chromium has been added in version 2.7.5.

Here are some simple examples:

```python
url = 'http://www.python.org/

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url + 'doc/')

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```
20.1.1. Browser Controller Objects

Browser controllers provide these methods which parallel three of the module-level convenience functions:

controller.**open**(url, new=0, autoraise=True)

Display url using the browser handled by this controller. If new is 1, a new browser window is opened if possible. If new is 2, a new browser page (“tab”) is opened if possible.

controller.**open_new**(url)

Open url in a new window of the browser handled by this controller, if possible, otherwise, open url in the only browser window. Alias open_new().

controller.**open_new_tab**(url)

Open url in a new page (“tab”) of the browser handled by this controller, if possible, otherwise equivalent to open_new().

*New in version 2.5.*

Footnotes

[1] Executables named here without a full path will be searched in the directories given in the **PATH** environment variable.
20.2. **cgi** — Common Gateway Interface support

**Source code:** Lib/cgi.py

Support module for Common Gateway Interface (CGI) scripts.

This module defines a number of utilities for use by CGI scripts written in Python.
20.2.1. Introduction

A CGI script is invoked by an HTTP server, usually to process user input submitted through an HTML `<FORM>` or `<ISINDEX>` element.

Most often, CGI scripts live in the server's special `cgi-bin` directory. The HTTP server places all sorts of information about the request (such as the client's hostname, the requested URL, the query string, and lots of other goodies) in the script's shell environment, executes the script, and sends the script's output back to the client.

The script's input is connected to the client too, and sometimes the form data is read this way; at other times the form data is passed via the "query string" part of the URL. This module is intended to take care of the different cases and provide a simpler interface to the Python script. It also provides a number of utilities that help in debugging scripts, and the latest addition is support for file uploads from a form (if your browser supports it).

The output of a CGI script should consist of two sections, separated by a blank line. The first section contains a number of headers, telling the client what kind of data is following. Python code to generate a minimal header section looks like this:

```python
print "Content-Type: text/html" # HTML is following
print # blank line, end of header
```

The second section is usually HTML, which allows the client software to display nicely formatted text with header, in-line images, etc. Here's Python code that prints a simple piece of HTML:

```python
print "<TITLE>CGI script output</TITLE>"
print "<H1>This is my first CGI script</H1>"
print "Hello, world!"
```
20.2.2. Using the cgi module

Begin by writing `import cgi`. Do not use `from cgi import *` — the module defines all sorts of names for its own use or for backward compatibility that you don’t want in your namespace.

When you write a new script, consider adding these lines:

```python
import cgitb

cgitb.enable()
```

This activates a special exception handler that will display detailed reports in the Web browser if any errors occur. If you’d rather not show the guts of your program to users of your script, you can have the reports saved to files instead, with code like this:

```python
import cgitb

cgitb.enable(display=0, logdir="/path/to/logdir")
```

It’s very helpful to use this feature during script development. The reports produced by `cgitb` provide information that can save you a lot of time in tracking down bugs. You can always remove the `cgitb` line later when you have tested your script and are confident that it works correctly.

To get at submitted form data, it’s best to use the `FieldStorage` class. The other classes defined in this module are provided mostly for backward compatibility. Instantiate it exactly once, without arguments. This reads the form contents from standard input or the environment (depending on the value of various environment variables set according to the CGI standard). Since it may consume standard input, it should be instantiated only once.

The `FieldStorage` instance can be indexed like a Python dictionary. It
allows membership testing with the in operator, and also supports the standard dictionary method keys() and the built-in function len(). Form fields containing empty strings are ignored and do not appear in the dictionary; to keep such values, provide a true value for the optional keep_blank_values keyword parameter when creating the FieldStorage instance.

For instance, the following code (which assumes that the Content-Type header and blank line have already been printed) checks that the fields name and addr are both set to a non-empty string:

```python
form = cgi.FieldStorage()
if "name" not in form or "addr" not in form:
    print "<H1>Error</H1>"
    print "Please fill in the name and addr fields."
    return
print "<p>name:“, form["name"].value
print "<p>addr:“, form["addr"].value
...further form processing here...
```

Here the fields, accessed through form[key], are themselves instances of FieldStorage (or MiniFieldStorage, depending on the form encoding). The value attribute of the instance yields the string value of the field. The getValue() method returns this string value directly; it also accepts an optional second argument as a default to return if the requested key is not present.

If the submitted form data contains more than one field with the same name, the object retrieved by form[key] is not a FieldStorage or MiniFieldStorage instance but a list of such instances. Similarly, in this situation, form.getValue(key) would return a list of strings. If you expect this possibility (when your HTML form contains multiple fields with the same name), use the getList() method, which always returns a list of values (so that you do not need to special-case the single item case). For example, this code concatenates any number
of username fields, separated by commas:

```python
value = form.getlist("username")
usernames = ",".join(value)
```

If a field represents an uploaded file, accessing the value via the `value` attribute or the `getvalue()` method reads the entire file in memory as a string. This may not be what you want. You can test for an uploaded file by testing either the `filename` attribute or the `file` attribute. You can then read the data at leisure from the `file` attribute:

```python
fileitem = form["userfile"]
if fileitem.file:
    # It's an uploaded file; count lines
    linecount = 0
    while 1:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1
```

If an error is encountered when obtaining the contents of an uploaded file (for example, when the user interrupts the form submission by clicking on a Back or Cancel button) the `done` attribute of the object for the field will be set to the value `-1`.

The file upload draft standard entertains the possibility of uploading multiple files from one field (using a recursive `multipart/*` encoding). When this occurs, the item will be a dictionary-like `FieldStorage` item. This can be determined by testing its `type` attribute, which should be `multipart/form-data` (or perhaps another MIME type matching `multipart/*`). In this case, it can be iterated over recursively just like the top-level form object.

When a form is submitted in the “old” format (as the query string or as a single data part of type `application/x-www-form-urlencoded`), the
items will actually be instances of the class `MiniFieldStorage`. In this case, the `list`, `file`, and `filename` attributes are always `None`.

A form submitted via POST that also has a query string will contain both `FieldStorage` and `MiniFieldStorage` items.
20.2.3. Higher Level Interface

New in version 2.2.

The previous section explains how to read CGI form data using the FieldStorage class. This section describes a higher level interface which was added to this class to allow one to do it in a more readable and intuitive way. The interface doesn't make the techniques described in previous sections obsolete — they are still useful to process file uploads efficiently, for example.

The interface consists of two simple methods. Using the methods you can process form data in a generic way, without the need to worry whether only one or more values were posted under one name.

In the previous section, you learned to write following code anytime you expected a user to post more than one value under one name:

```python
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

This situation is common for example when a form contains a group of multiple checkboxes with the same name:

```html
<input type="checkbox" name="item" value="1" />
<input type="checkbox" name="item" value="2" />
```

In most situations, however, there's only one form control with a particular name in a form and then you expect and need only one value associated with this name. So you write a script containing for example this code:
The problem with the code is that you should never expect that a client will provide valid input to your scripts. For example, if a curious user appends another `user=foo` pair to the query string, then the script would crash, because in this situation the `getvalue("user")` method call returns a list instead of a string. Calling the `upper()` method on a list is not valid (since lists do not have a method of this name) and results in an `AttributeError` exception.

Therefore, the appropriate way to read form data values was to always use the code which checks whether the obtained value is a single value or a list of values. That's annoying and leads to less readable scripts.

A more convenient approach is to use the methods `getfirst()` and `getlist()` provided by this higher level interface.

**FieldStorage.getfirst(name[, default])**

This method always returns only one value associated with form field `name`. The method returns only the first value in case that more values were posted under such name. Please note that the order in which the values are received may vary from browser to browser and should not be counted on. [1] If no such form field or value exists then the method returns the value specified by the optional parameter `default`. This parameter defaults to `None` if not specified.

**FieldStorage.getlist(name)**

This method always returns a list of values associated with form field `name`. The method returns an empty list if no such form field or value exists for `name`. It returns a list consisting of one item if only one such value exists.
Using these methods you can write nice compact code:

```python
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", ").upper()  # This way it's safe
for item in form.getlist("item"):
    do_something(item)
```
20.2.4. Old classes

*Deprecated since version 2.6:* These classes, present in earlier versions of the `cgi` module, are still supported for backward compatibility. New applications should use the `FieldStorage` class.

**SvFormContentDict** stores single value form content as dictionary; it assumes each field name occurs in the form only once.

**FormContentDict** stores multiple value form content as a dictionary (the form items are lists of values). Useful if your form contains multiple fields with the same name.

Other classes (**FormContent**, **InterpFormContentDict**) are present for backwards compatibility with really old applications only.
20.2.5. Functions

These are useful if you want more control, or if you want to employ some of the algorithms implemented in this module in other circumstances.

cgi.parse(fp[, environ[, keep_blank_values[, strict_parsing]]]])

Parse a query in the environment or from a file (the file defaults to sys.stdin and environment defaults to os.environ). The keep_blank_values and strict_parsing parameters are passed to urlparse.parse_qs() unchanged.

cgi.parse_qs(qs[, keep_blank_values[, strict_parsing]])

This function is deprecated in this module. Use urlparse.parse_qs() instead. It is maintained here only for backward compatibility.

cgi.parse_qsl(qs[, keep_blank_values[, strict_parsing]])

This function is deprecated in this module. Use urlparse.parse_qsl() instead. It is maintained here only for backward compatibility.

cgi.parse_multipart(fp, pdict)

Parse input of type multipart/form-data (for file uploads). Arguments are fp for the input file and pdict for a dictionary containing other parameters in the Content-Type header.

Returns a dictionary just like urlparse.parse_qs() keys are the field names, each value is a list of values for that field. This is easy to use but not much good if you are expecting megabytes to be uploaded — in that case, use the FieldStorage class instead which is much more flexible.
Note that this does not parse nested multipart parts — use FieldStorage for that.

cgi.parse_header(string)
Parse a MIME header (such as Content-Type) into a main value and a dictionary of parameters.

cgi.test()
Robust test CGI script, usable as main program. Writes minimal HTTP headers and formats all information provided to the script in HTML form.

cgi.print_environ()
Format the shell environment in HTML.

cgi.print_form(form)
Format a form in HTML.

cgi.print_directory()
Format the current directory in HTML.

cgi.print_environ_usage()
Print a list of useful (used by CGI) environment variables in HTML.

cgi.escape(s[, quote])
Convert the characters '&', '<' and '>' in string s to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. If the optional flag quote is true, the quotation mark character (") is also translated; this helps for inclusion in an HTML attribute value delimited by double quotes, as in <a href="...">. Note that single quotes are never translated.

If the value to be quoted might include single- or double-quote
characters, or both, consider using the `quoteattr()` function in the `xml.sax.saxutils` module instead.
20.2.6. Caring about security

There's one important rule: if you invoke an external program (via the `os.system()` or `os.popen()` functions, or others with similar functionality), make very sure you don't pass arbitrary strings received from the client to the shell. This is a well-known security hole whereby clever hackers anywhere on the Web can exploit a gullible CGI script to invoke arbitrary shell commands. Even parts of the URL or field names cannot be trusted, since the request doesn't have to come from your form!

To be on the safe side, if you must pass a string gotten from a form to a shell command, you should make sure the string contains only alphanumeric characters, dashes, underscores, and periods.
20.2.7. Installing your CGI script on a Unix system

Read the documentation for your HTTP server and check with your local system administrator to find the directory where CGI scripts should be installed; usually this is in a directory `cgi-bin` in the server tree.

Make sure that your script is readable and executable by “others”; the Unix file mode should be `0755` octal (use `chmod 0755 filename`). Make sure that the first line of the script contains `#!` starting in column 1 followed by the pathname of the Python interpreter, for instance:

```
#!/usr/local/bin/python
```

Make sure the Python interpreter exists and is executable by “others”.

Make sure that any files your script needs to read or write are readable or writable, respectively, by “others” — their mode should be `0644` for readable and `0666` for writable. This is because, for security reasons, the HTTP server executes your script as user “nobody”, without any special privileges. It can only read (write, execute) files that everybody can read (write, execute). The current directory at execution time is also different (it is usually the server’s `cgi-bin` directory) and the set of environment variables is also different from what you get when you log in. In particular, don’t count on the shell’s search path for executables (`PATH`) or the Python module search path (`PYTHONPATH`) to be set to anything interesting.

If you need to load modules from a directory which is not on Python’s default module search path, you can change the path in your script,
before importing other modules. For example:

```python
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

(This way, the directory inserted last will be searched first!)

Instructions for non-Unix systems will vary; check your HTTP server’s documentation (it will usually have a section on CGI scripts).
20.2.8. Testing your CGI script

Unfortunately, a CGI script will generally not run when you try it from the command line, and a script that works perfectly from the command line may fail mysteriously when run from the server. There’s one reason why you should still test your script from the command line: if it contains a syntax error, the Python interpreter won’t execute it at all, and the HTTP server will most likely send a cryptic error to the client.

Assuming your script has no syntax errors, yet it does not work, you have no choice but to read the next section.
20.2.9. Debugging CGI scripts

First of all, check for trivial installation errors — reading the section above on installing your CGI script carefully can save you a lot of time. If you wonder whether you have understood the installation procedure correctly, try installing a copy of this module file (cgi.py) as a CGI script. When invoked as a script, the file will dump its environment and the contents of the form in HTML form. Give it the right mode etc, and send it a request. If it's installed in the standard cgi-bin directory, it should be possible to send it a request by entering a URL into your browser of the form:

http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&amp;addr=At+Home

If this gives an error of type 404, the server cannot find the script – perhaps you need to install it in a different directory. If it gives another error, there's an installation problem that you should fix before trying to go any further. If you get a nicely formatted listing of the environment and form content (in this example, the fields should be listed as “addr” with value “At Home” and “name” with value “Joe Blow”), the cgi.py script has been installed correctly. If you follow the same procedure for your own script, you should now be able to debug it.

The next step could be to call the cgi module's test() function from your script: replace its main code with the single statement

cgi.test()

This should produce the same results as those gotten from installing the cgi.py file itself.

When an ordinary Python script raises an unhandled exception (for
whatever reason: of a typo in a module name, a file that can’t be opened, etc.), the Python interpreter prints a nice traceback and exits. While the Python interpreter will still do this when your CGI script raises an exception, most likely the traceback will end up in one of the HTTP server’s log files, or be discarded altogether.

Fortunately, once you have managed to get your script to execute some code, you can easily send tracebacks to the Web browser using the cgitb module. If you haven’t done so already, just add the lines:

```
import cgitb

cgitb.enable()
```

to the top of your script. Then try running it again; when a problem occurs, you should see a detailed report that will likely make apparent the cause of the crash.

If you suspect that there may be a problem in importing the cgitb module, you can use an even more robust approach (which only uses built-in modules):

```
import sys
sys.stderr = sys.stdout
print "Content-Type: text/plain"
print
...your code here...
```

This relies on the Python interpreter to print the traceback. The content type of the output is set to plain text, which disables all HTML processing. If your script works, the raw HTML will be displayed by your client. If it raises an exception, most likely after the first two lines have been printed, a traceback will be displayed. Because no HTML interpretation is going on, the traceback will be readable.
20.2.10. Common problems and solutions

- Most HTTP servers buffer the output from CGI scripts until the script is completed. This means that it is not possible to display a progress report on the client's display while the script is running.
- Check the installation instructions above.
- Check the HTTP server’s log files. (tail -f logfile in a separate window may be useful!)
- Always check a script for syntax errors first, by doing something like `python script.py`.
- If your script does not have any syntax errors, try adding `import cgitb; cgitb.enable()` to the top of the script.
- When invoking external programs, make sure they can be found. Usually, this means using absolute path names — PATH is usually not set to a very useful value in a CGI script.
- When reading or writing external files, make sure they can be read or written by the userid under which your CGI script will be running: this is typically the userid under which the web server is running, or some explicitly specified userid for a web server’s suexec feature.
- Don’t try to give a CGI script a set-uid mode. This doesn’t work on most systems, and is a security liability as well.

Footnotes

Note that some recent versions of the HTML specification do state what order the field values should be supplied in, but knowing whether a request was received from a conforming browser, or even from a browser at all, is tedious and error-prone.
20. Internet Protocols and Support
20.3. cgitb — Traceback manager for CGI scripts

New in version 2.2.

The cgitb module provides a special exception handler for Python scripts. (Its name is a bit misleading. It was originally designed to display extensive traceback information in HTML for CGI scripts. It was later generalized to also display this information in plain text.) After this module is activated, if an uncaught exception occurs, a detailed, formatted report will be displayed. The report includes a traceback showing excerpts of the source code for each level, as well as the values of the arguments and local variables to currently running functions, to help you debug the problem. Optionally, you can save this information to a file instead of sending it to the browser.

To enable this feature, simply add this to the top of your CGI script:

```python
import cgitb

cgitb.enable()
```

The options to the `enable()` function control whether the report is displayed in the browser and whether the report is logged to a file for later analysis.

```python
cgitb.enable([display[, logdir[, context[, format]]]])
```

This function causes the cgitb module to take over the interpreter’s default handling for exceptions by setting the value of `sys.excepthook`.

The optional argument `display` defaults to 1 and can be set to 0 to suppress sending the traceback to the browser. If the
argument \textit{logdir} is present, the traceback reports are written to files. The value of \textit{logdir} should be a directory where these files will be placed. The optional argument \textit{context} is the number of lines of context to display around the current line of source code in the traceback; this defaults to 5. If the optional argument \textit{format} is "html", the output is formatted as HTML. Any other value forces plain text output. The default value is "html".

\texttt{cgitb.\texttt{handler}([\textit{info}])}

This function handles an exception using the default settings (that is, show a report in the browser, but don't log to a file). This can be used when you've caught an exception and want to report it using \texttt{cgitb}. The optional \textit{info} argument should be a 3-tuple containing an exception type, exception value, and traceback object, exactly like the tuple returned by \texttt{sys.exc_info()}. If the \textit{info} argument is not supplied, the current exception is obtained from \texttt{sys.exc_info()}. 

20.4. wsgiref — WSGI Utilities and Reference Implementation

New in version 2.5.

The Web Server Gateway Interface (WSGI) is a standard interface between web server software and web applications written in Python. Having a standard interface makes it easy to use an application that supports WSGI with a number of different web servers.

Only authors of web servers and programming frameworks need to know every detail and corner case of the WSGI design. You don’t need to understand every detail of WSGI just to install a WSGI application or to write a web application using an existing framework.

wsgiref is a reference implementation of the WSGI specification that can be used to add WSGI support to a web server or framework. It provides utilities for manipulating WSGI environment variables and response headers, base classes for implementing WSGI servers, a demo HTTP server that serves WSGI applications, and a validation tool that checks WSGI servers and applications for conformance to the WSGI specification (PEP 333).

See https://wsgi.readthedocs.org/ for more information about WSGI, and links to tutorials and other resources.
This module provides a variety of utility functions for working with WSGI environments. A WSGI environment is a dictionary containing HTTP request variables as described in PEP 333. All of the functions taking an `environ` parameter expect a WSGI-compliant dictionary to be supplied; please see PEP 333 for a detailed specification.

**wsgiref.util.guess_scheme(environ)**

Return a guess for whether `wsgi.url_scheme` should be “http” or “https”, by checking for a `HTTPS` environment variable in the `environ` dictionary. The return value is a string.

This function is useful when creating a gateway that wraps CGI or a CGI-like protocol such as FastCGI. Typically, servers providing such protocols will include a `HTTPS` variable with a value of “1” “yes”, or “on” when a request is received via SSL. So, this function returns “https” if such a value is found, and “http” otherwise.

**wsgiref.util.request_uri(environ, include_query=True)**

Return the full request URI, optionally including the query string, using the algorithm found in the “URL Reconstruction” section of PEP 333. If `include_query` is false, the query string is not included in the resulting URI.

**wsgiref.util.application_uri(environ)**

Similar to `request_uri()`, except that the `PATH_INFO` and `QUERY_STRING` variables are ignored. The result is the base URI of the application object addressed by the request.

**wsgiref.util.shift_path_info(environ)**
Shift a single name from `PATH_INFO` to `SCRIPT_NAME` and return the name. The `environ` dictionary is *modified* in-place; use a copy if you need to keep the original `PATH_INFO` or `SCRIPT_NAME` intact.

If there are no remaining path segments in `PATH_INFO`, `None` is returned.

Typically, this routine is used to process each portion of a request URI path, for example to treat the path as a series of dictionary keys. This routine modifies the passed-in environment to make it suitable for invoking another WSGI application that is located at the target URI. For example, if there is a WSGI application at `/foo`, and the request URI path is `/foo/bar/baz`, and the WSGI application at `/foo` calls `shift_path_info()`, it will receive the string “bar”, and the environment will be updated to be suitable for passing to a WSGI application at `/foo/bar`. That is, `SCRIPT_NAME` will change from `/foo` to `/foo/bar`, and `PATH_INFO` will change from `/bar/baz` to `/baz`.

When `PATH_INFO` is just a “/”, this routine returns an empty string and appends a trailing slash to `SCRIPT_NAME`, even though empty path segments are normally ignored, and `SCRIPT_NAME` doesn’t normally end in a slash. This is intentional behavior, to ensure that an application can tell the difference between URIs ending in `/x` from ones ending in `/x/` when using this routine to do object traversal.

```python
wsgiref.util.setup_testing_defaults(environ)
```

Update `environ` with trivial defaults for testing purposes.

This routine adds various parameters required for WSGI, including `HTTP_HOST`, `SERVER_NAME`, `SERVER_PORT`, `REQUEST_METHOD`, `SCRIPT_NAME`, `PATH_INFO`, and all of the `PEP 333`-defined `wsgi.*` variables. It only supplies default values, and does not replace
any existing settings for these variables.

This routine is intended to make it easier for unit tests of WSGI servers and applications to set up dummy environments. It should NOT be used by actual WSGI servers or applications, since the data is fake!

Example usage:

```python
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It's going to print
# environment dictionary after being updated by setup_testing_defaults

def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain')]

    start_response(status, headers)
    ret = ['%s: %s
' % (key, value)
           for key, value in environ.iteritems()]
    return ret

httpd = make_server('', 8000, simple_app)
print "Serving on port 8000..."
httpd.serve_forever()
```

In addition to the environment functions above, the `wsgiref.util` module also provides these miscellaneous utilities:

`wsgiref.util.is_hop_by_hop(header_name)`

Return `true` if ‘header_name’ is an HTTP/1.1 “Hop-by-Hop” header, as defined by RFC 2616.

`class wsgiref.util.FileWrapper(filelike, blksize=8192)`

A wrapper to convert a file-like object to an iterator. The resulting objects support both `__getitem__()` and `__iter__()` iteration
styles, for compatibility with Python 2.1 and Jython. As the object is iterated over, the optional \texttt{blksize} parameter will be repeatedly passed to the \texttt{filelike} object's \texttt{read()} method to obtain strings to yield. When \texttt{read()} returns an empty string, iteration is ended and is not resumable.

If \texttt{filelike} has a \texttt{close()} method, the returned object will also have a \texttt{close()} method, and it will invoke the \texttt{filelike} object's \texttt{close()} method when called.

Example usage:

```python
from StringIO import StringIO
from wsgiref.util import FileWrapper

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object"*10)
wrapper = FileWrapper(filelike, blksize=5)

for chunk in wrapper:
    print chunk
```
This module provides a single class, `Headers`, for convenient manipulation of WSGI response headers using a mapping-like interface.

```python
class wsgiref.headers.Headers(headers)
    Create a mapping-like object wrapping *headers*, which must be a list of header name/value tuples as described in PEP 333. Any changes made to the new `Headers` object will directly update the *headers* list it was created with.
```

`Headers` objects support typical mapping operations including `__getitem__()`, `get()`, `__setitem__()`, `setdefault()`, `__delitem__()`, `__contains__()`, and `has_key()`. For each of these methods, the key is the header name (treated case-insensitively), and the value is the first value associated with that header name. Setting a header deletes any existing values for that header, then adds a new value at the end of the wrapped header list. Headers’ existing order is generally maintained, with new headers added to the end of the wrapped list.

Unlike a dictionary, `Headers` objects do not raise an error when you try to get or delete a key that isn’t in the wrapped header list. Getting a nonexistent header just returns `None`, and deleting a nonexistent header does nothing.

`Headers` objects also support `keys()`, `values()`, and `items()` methods. The lists returned by `keys()` and `items()` can include the same key more than once if there is a multi-valued header. The `len()` of a `Headers` object is the same as the length of its
items(), which is the same as the length of the wrapped header list. In fact, the items() method just returns a copy of the wrapped header list.

Calling str() on a Headers object returns a formatted string suitable for transmission as HTTP response headers. Each header is placed on a line with its value, separated by a colon and a space. Each line is terminated by a carriage return and line feed, and the string is terminated with a blank line.

In addition to their mapping interface and formatting features, Headers objects also have the following methods for querying and adding multi-valued headers, and for adding headers with MIME parameters:

get_all(name)
Return a list of all the values for the named header.

The returned list will be sorted in the order they appeared in the original header list or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list. If no fields exist with the given name, returns an empty list.

add_header(name, value, **_params)
Add a (possibly multi-valued) header, with optional MIME parameters specified via keyword arguments.

name is the header field to add. Keyword arguments can be used to set MIME parameters for the header field. Each parameter must be a string or None. Underscores in parameter names are converted to dashes, since dashes are illegal in Python identifiers, but many MIME parameter names include dashes. If the parameter value is a string, it is added to the header value parameters in the form name="value". If it
is None, only the parameter name is added. (This is used for MIME parameters without a value.) Example usage:

```python
h.add_header('content-disposition', 'attachment', filename=None)
```

The above will add a header that looks like this:

```
Content-Disposition: attachment; filename="bud.gif"
```
20.4.3. wsgiref.simple_server – a simple WSGI HTTP server

This module implements a simple HTTP server (based on BaseHTTPServer) that serves WSGI applications. Each server instance serves a single WSGI application on a given host and port. If you want to serve multiple applications on a single host and port, you should create a WSGI application that parses PATH_INFO to select which application to invoke for each request. (E.g., using the shift_path_info() function from wsgiref.util.)

wsgiref.simple_server.make_server(host, port, app, server_class=WSGIServer, handler_class=WSGIRequestHandler)

Create a new WSGI server listening on host and port, accepting connections for app. The return value is an instance of the supplied server_class, and will process requests using the specified handler_class. app must be a WSGI application object, as defined by PEP 333.

Example usage:

```python
from wsgiref.simple_server import make_server, demo_app

httpd = make_server('', 8000, demo_app)
print "Serving HTTP on port 8000..."

# Respond to requests until process is killed
httpd.serve_forever()

# Alternative: serve one request, then exit
httpd.handle_request()
```

wsgiref.simple_server.demo_app(environ, start_response)

This function is a small but complete WSGI application that returns a text page containing the message “Hello world!” and a
list of the key/value pairs provided in the `environ` parameter. It’s useful for verifying that a WSGI server (such as `wsgiref.simple_server`) is able to run a simple WSGI application correctly.

```python
class wsgiref.simple_server.WSGIServer(server_address, RequestHandlerClass)
    Create a WSGIServer instance. server_address should be a (host,port) tuple, and RequestHandlerClass should be the subclass of `BaseHTTPServer.BaseHTTPRequestHandler` that will be used to process requests.
```

You do not normally need to call this constructor, as the `make_server()` function can handle all the details for you.

```python
WSGIServer is a subclass of `BaseHTTPServer.HTTPServer`, so all of its methods (such as `serve_forever()` and `handle_request()`) are available. WSGIServer also provides these WSGI-specific methods:

```python
    set_app(application)
        Sets the callable application as the WSGI application that will receive requests.

    get_app()
        Returns the currently-set application callable.
```

Normally, however, you do not need to use these additional methods, as `set_app()` is normally called by `make_server()`, and the `get_app()` exists mainly for the benefit of request handler instances.

```python
class wsgiref.simple_server.WSGIRequestHandler(request, client_address, server)
    Create an HTTP handler for the given request (i.e. a socket),
client_address (a (host, port) tuple), and server (WSGIServer instance).

You do not need to create instances of this class directly; they are automatically created as needed by WSGIServer objects. You can, however, subclass this class and supply it as a handler_class to the make_server() function. Some possibly relevant methods for overriding in subclasses:

get_environ()
Returns a dictionary containing the WSGI environment for a request. The default implementation copies the contents of the WSGIServer object’s base_environ dictionary attribute and then adds various headers derived from the HTTP request. Each call to this method should return a new dictionary containing all of the relevant CGI environment variables as specified in PEP 333.

get_stderr()
Return the object that should be used as the wsgi.errors stream. The default implementation just returns sys.stderr.

handle()
Process the HTTP request. The default implementation creates a handler instance using a wsgiref.handlers class to implement the actual WSGI application interface.
When creating new WSGI application objects, frameworks, servers, or middleware, it can be useful to validate the new code's conformance using `wsgiref.validate`. This module provides a function that creates WSGI application objects that validate communications between a WSGI server or gateway and a WSGI application object, to check both sides for protocol conformance.

Note that this utility does not guarantee complete PEP 333 compliance; an absence of errors from this module does not necessarily mean that errors do not exist. However, if this module does produce an error, then it is virtually certain that either the server or application is not 100% compliant.

This module is based on the `paste.lint` module from Ian Bicking's “Python Paste” library.

`wsgiref.validate.validator(application)`

Wrap `application` and return a new WSGI application object. The returned application will forward all requests to the original `application`, and will check that both the `application` and the server invoking it are conforming to the WSGI specification and to RFC 2616.

Any detected nonconformance results in an `AssertionError` being raised; note, however, that how these errors are handled is server-dependent. For example, `wsgiref.simple_server` and other servers based on `wsgiref.handlers` (that don't override the error handling methods to do something else) will simply output a message that an error has occurred, and dump the traceback to `sys.stderr` or some other error stream.
This wrapper may also generate output using the `warnings` module to indicate behaviors that are questionable but which may not actually be prohibited by PEP 333. Unless they are suppressed using Python command-line options or the `warnings` API, any such warnings will be written to `sys.stderr` (not `wsgi.errors`, unless they happen to be the same object).

Example usage:

```python
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not compliant to the standard, so the validator is going to break
def simple_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

    # This is going to break because we need to return a list, and the validator is going to inform us
    return 'Hello World'

# This is the application wrapped in a validator
validator_app = validator(simple_app)

httpd = make_server('', 8000, validator_app)
print "Listening on port 8000...."
httpd.serve_forever()
```
20.4.5. wsgiref.handlers – server/gateway base classes

This module provides base handler classes for implementing WSGI servers and gateways. These base classes handle most of the work of communicating with a WSGI application, as long as they are given a CGI-like environment, along with input, output, and error streams.

```
class wsgiref.handlers.CGIHandler
    CGI-based invocation via sys.stdin, sys.stdout, sys.stderr and os.environ. This is useful when you have a WSGI application and want to run it as a CGI script. Simply invoke CGIHandler().run(app), where app is the WSGI application object you wish to invoke.
```

This class is a subclass of BaseCGIHandler that sets wsgi.run_once to true, wsgi.multithread to false, and wsgi.multiprocess to true, and always uses sys and os to obtain the necessary CGI streams and environment.

```
class wsgiref.handlers.BaseCGIHandler(stdin, stdout, stderr, environ, multithread=True, multiprocess=False)
    Similar to CGIHandler, but instead of using the sys and os modules, the CGI environment and I/O streams are specified explicitly. The multithread and multiprocess values are used to set the wsgi.multithread and wsgi.multiprocess flags for any applications run by the handler instance.
```

This class is a subclass of SimpleHandler intended for use with software other than HTTP “origin servers”. If you are writing a gateway protocol implementation (such as CGI, FastCGI, SCGI, etc.) that uses a Status: header to send an HTTP status, you
probably want to subclass this instead of SimpleHandler.

```
class wsgiref.handlers.SimpleHandler(stdin, stdout, stderr, environ, multithread=True, multiprocessing=False)
```

Similar to BaseCGIHandler, but designed for use with HTTP origin servers. If you are writing an HTTP server implementation, you will probably want to subclass this instead of BaseCGIHandler.

This class is a subclass of BaseHandler. It overrides the __init__(), get_stdin(), get_stderr(), add_cgi_vars(), _write(), and _flush() methods to support explicitly setting the environment and streams via the constructor. The supplied environment and streams are stored in the stdin, stdout, stderr, and environ attributes.

```
class wsgiref.handlers.BaseHandler
```

This is an abstract base class for running WSGI applications. Each instance will handle a single HTTP request, although in principle you could create a subclass that was reusable for multiple requests.

BaseHandler instances have only one method intended for external use:

```
run(app)
```

Run the specified WSGI application, app.

All of the other BaseHandler methods are invoked by this method in the process of running the application, and thus exist primarily to allow customizing the process.

The following methods MUST be overridden in a subclass:

```
_write(data)
```

Buffer the string data for transmission to the client. It's okay if
this method actually transmits the data; BaseHandler just separates write and flush operations for greater efficiency when the underlying system actually has such a distinction.

_flush()
Force buffered data to be transmitted to the client. It’s okay if this method is a no-op (i.e., if _write() actually sends the data).

get_stdin()
Return an input stream object suitable for use as the wsgi.input of the request currently being processed.

get_stderr()
Return an output stream object suitable for use as the wsgi.errors of the request currently being processed.

add_cgi_vars()
Insert CGI variables for the current request into the environ attribute.

Here are some other methods and attributes you may wish to override. This list is only a summary, however, and does not include every method that can be overridden. You should consult the docstrings and source code for additional information before attempting to create a customized BaseHandler subclass.

Attributes and methods for customizing the WSGI environment:

wsgi_multithread
The value to be used for the wsgi.multithread environment variable. It defaults to true in BaseHandler, but may have a different default (or be set by the constructor) in the other subclasses.
wsgi_multiprocess
The value to be used for the `wsgi.multiprocess` environment variable. It defaults to true in `BaseHandler`, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_run_once
The value to be used for the `wsgi.run_once` environment variable. It defaults to false in `BaseHandler`, but `CGIHandler` sets it to true by default.

os_environ
The default environment variables to be included in every request’s WSGI environment. By default, this is a copy of `os.environ` at the time that `wsgiref.handlers` was imported, but subclasses can either create their own at the class or instance level. Note that the dictionary should be considered read-only, since the default value is shared between multiple classes and instances.

server_software
If the `origin_server` attribute is set, this attribute’s value is used to set the default `SERVER_SOFTWARE` WSGI environment variable, and also to set a default `Server:` header in HTTP responses. It is ignored for handlers (such as `BaseCGIHandler` and `CGIHandler`) that are not HTTP origin servers.

get_scheme()
Return the URL scheme being used for the current request. The default implementation uses the `guess_scheme()` function from `wsgiref.util` to guess whether the scheme should be “http” or “https”, based on the current request’s `environ` variables.
**setup_environ()**
Set the `environ` attribute to a fully-populated WSGI environment. The default implementation uses all of the above methods and attributes, plus the `get_stdin()`, `get_stderr()`, and `add_cgi_vars()` methods and the `wsgi_file_wrapper` attribute. It also inserts a `SERVER_SOFTWARE` key if not present, as long as the `origin_server` attribute is a true value and the `server_software` attribute is set.

Methods and attributes for customizing exception handling:

**log_exception(exc_info)**
Log the `exc_info` tuple in the server log. `exc_info` is a `(type, value, traceback)` tuple. The default implementation simply writes the traceback to the request’s `wsgi.errors` stream and flushes it. Subclasses can override this method to change the format or retarget the output, mail the traceback to an administrator, or whatever other action may be deemed suitable.

**traceback_limit**
The maximum number of frames to include in tracebacks output by the default `log_exception()` method. If `None`, all frames are included.

**error_output(environ, start_response)**
This method is a WSGI application to generate an error page for the user. It is only invoked if an error occurs before headers are sent to the client.

This method can access the current error information using `sys.exc_info()`, and should pass that information to `start_response` when calling it (as described in the “Error Handling” section of PEP 333).
The default implementation just uses the `error_status`, `error_headers`, and `error_body` attributes to generate an output page. Subclasses can override this to produce more dynamic error output.

Note, however, that it’s not recommended from a security perspective to spit out diagnostics to any old user; ideally, you should have to do something special to enable diagnostic output, which is why the default implementation doesn’t include any.

**error_status**

The HTTP status used for error responses. This should be a status string as defined in PEP 333; it defaults to a 500 code and message.

**error_headers**

The HTTP headers used for error responses. This should be a list of WSGI response headers `((name, value) tuples)`, as described in PEP 333. The default list just sets the content type to `text/plain`.

**error_body**

The error response body. This should be an HTTP response body string. It defaults to the plain text, “A server error occurred. Please contact the administrator.”

Methods and attributes for PEP 333’s “Optional Platform-Specific File Handling” feature:

**wsgi_file_wrapper**

A `wsgi.file_wrapper` factory, or `None`. The default value of this attribute is the `FileWrapper` class from `wsgiref.util`.

**sendfile()**
Override to implement platform-specific file transmission. This method is called only if the application's return value is an instance of the class specified by the `wsgi_file_wrapper` attribute. It should return a true value if it was able to successfully transmit the file, so that the default transmission code will not be executed. The default implementation of this method just returns a false value.

Miscellaneous methods and attributes:

**origin_server**

This attribute should be set to a true value if the handler's `_write()` and `_flush()` are being used to communicate directly to the client, rather than via a CGI-like gateway protocol that wants the HTTP status in a special `Status:` header.

This attribute's default value is true in `BaseHandler`, but false in `BaseCGIHandler` and `CGIHandler`.

**http_version**

If `origin_server` is true, this string attribute is used to set the HTTP version of the response set to the client. It defaults to "1.0".
This is a working “Hello World” WSGI application:

```python
from wsgiref.simple_server import make_server

def hello_world_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

    return ['Hello World']

httpd = make_server('', 8000, hello_world_app)
print "Serving on port 8000..."

httpd.serve_forever()
```
20.5. `urllib` — Open arbitrary resources by URL

**Note:** The `urllib` module has been split into parts and renamed in Python 3 to `urllib.request`, `urllib.parse`, and `urllib.error`. The `2to3` tool will automatically adapt imports when converting your sources to Python 3. Also note that the `urllib.request.urlopen()` function in Python 3 is equivalent to `urllib2.urlopen()` and that `urllib.urlopen()` has been removed.

This module provides a high-level interface for fetching data across the World Wide Web. In particular, the `urlopen()` function is similar to the built-in function `open()`, but accepts Universal Resource Locators (URLs) instead of filenames. Some restrictions apply — it can only open URLs for reading, and no seek operations are available.

**See also:** The Requests package is recommended for a higher-level HTTP client interface.

**Warning:** When opening HTTPS URLs, it does not attempt to validate the server certificate. Use at your own risk!
20.5.1. High-level interface

urllib.urlopen(url[, data[, proxies[, context]]])

Open a network object denoted by a URL for reading. If the URL does not have a scheme identifier, or if it has file: as its scheme identifier, this opens a local file (without universal newlines); otherwise it opens a socket to a server somewhere on the network. If the connection cannot be made the IOError exception is raised. If all went well, a file-like object is returned. This supports the following methods: read(), readline(), readlines(), fileno(), close(), info(), getcode() and geturl(). It also has proper support for the iterator protocol. One caveat: the read() method, if the size argument is omitted or negative, may not read until the end of the data stream; there is no good way to determine that the entire stream from a socket has been read in the general case.

Except for the info(), getcode() and geturl() methods, these methods have the same interface as for file objects — see section File Objects in this manual. (It is not a built-in file object, however, so it can’t be used at those few places where a true built-in file object is required.)

The info() method returns an instance of the class mimetools.Message containing meta-information associated with the URL. When the method is HTTP, these headers are those returned by the server at the head of the retrieved HTML page (including Content-Length and Content-Type). When the method is FTP, a Content-Length header will be present if (as is now usual) the server passed back a file length in response to the FTP retrieval request. A Content-Type header will be present if the MIME type can be guessed. When the method is local-file,
returned headers will include a Date representing the file’s last-modified time, a Content-Length giving file size, and a Content-Type containing a guess at the file’s type. See also the description of the `mimetypes` module.

The `geturl()` method returns the real URL of the page. In some cases, the HTTP server redirects a client to another URL. The `urlopen()` function handles this transparently, but in some cases the caller needs to know which URL the client was redirected to. The `geturl()` method can be used to get at this redirected URL.

The `getcode()` method returns the HTTP status code that was sent with the response, or `None` if the URL is no HTTP URL.

If the `url` uses the `http:` scheme identifier, the optional `data` argument may be given to specify a `POST` request (normally the request type is `GET`). The `data` argument must be in standard `application/x-www-form-urlencoded` format; see the `urlencode()` function below.

The `urlopen()` function works transparently with proxies which do not require authentication. In a Unix or Windows environment, set the `http_proxy`, or `ftp_proxy` environment variables to a URL that identifies the proxy server before starting the Python interpreter. For example (the '%' is the command prompt):

```
% http_proxy="http://www.someproxy.com:3128"
% export http_proxy
% python
...
```

The `no_proxy` environment variable can be used to specify hosts which shouldn’t be reached via proxy; if set, it should be a comma-separated list of hostname suffixes, optionally with `:port`
appended, for example `cern.ch,ncsa.uiuc.edu,some.host:8080`.

In a Windows environment, if no proxy environment variables are set, proxy settings are obtained from the registry’s Internet Settings section.

In a Mac OS X environment, `urlopen()` will retrieve proxy information from the OS X System Configuration Framework, which can be managed with Network System Preferences panel.

Alternatively, the optional `proxies` argument may be used to explicitly specify proxies. It must be a dictionary mapping scheme names to proxy URLs, where an empty dictionary causes no proxies to be used, and `None` (the default value) causes environmental proxy settings to be used as discussed above. For example:

```python
# Use http://www.someproxy.com:3128 for HTTP proxying
proxies = {'http': 'http://www.someproxy.com:3128'}
filehandle = urllib.urlopen(some_url, proxies=proxies)
# Don't use any proxies
filehandle = urllib.urlopen(some_url, proxies={})
# Use proxies from environment - both versions are equivalent
filehandle = urllib.urlopen(some_url, proxies=None)
filehandle = urllib.urlopen(some_url)
```

Proxies which require authentication for use are not currently supported; this is considered an implementation limitation.

The `context` parameter may be set to a `ssl.SSLContext` instance to configure the SSL settings that are used if `urlopen()` makes a HTTPS connection.

*Changed in version 2.3:* Added the `proxies` support.

*Changed in version 2.6:* Added `getcode()` to returned object and
support for the `no_proxy` environment variable.

*Changed in version 2.7.9:* The `context` parameter was added.

*Deprecation notice:*  
*Deprecation since version 2.6:* The `urlopen()` function has been removed in Python 3 in favor of `urllib2.urlopen()`.

```python
urllib.urlopen(url[, filename[, reporthook[, data]]])
```

Copy a network object denoted by a URL to a local file, if necessary. If the URL points to a local file, or a valid cached copy of the object exists, the object is not copied. Return a tuple `(filename, headers)` where `filename` is the local file name under which the object can be found, and `headers` is whatever the `info()` method of the object returned by `urlopen()` returned (for a remote object, possibly cached). Exceptions are the same as for `urlopen()`.

The second argument, if present, specifies the file location to copy to (if absent, the location will be a tempfile with a generated name). The third argument, if present, is a hook function that will be called once on establishment of the network connection and once after each block read thereafter. The hook will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be `-1` on older FTP servers which do not return a file size in response to a retrieval request.

If the `url` uses the `http:` scheme identifier, the optional `data` argument may be given to specify a `POST` request (normally the request type is `GET`). The `data` argument must in standard `application/x-www-form-urlencoded` format; see the `urlencode()` function below.
Changed in version 2.5: `urlretrieve()` will raise `ContentTooShortError` when it detects that the amount of data available was less than the expected amount (which is the size reported by a `Content-Length` header). This can occur, for example, when the download is interrupted.

The `Content-Length` is treated as a lower bound: if there’s more data to read, `urlretrieve()` reads more data, but if less data is available, it raises the exception.

You can still retrieve the downloaded data in this case, it is stored in the `content` attribute of the exception instance.

If no `Content-Length` header was supplied, `urlretrieve()` can not check the size of the data it has downloaded, and just returns it. In this case you just have to assume that the download was successful.

`urllib._urlopener`

The public functions `urlopen()` and `urlretrieve()` create an instance of the `FancyURLopener` class and use it to perform their requested actions. To override this functionality, programmers can create a subclass of `URLopener` or `FancyURLopener`, then assign an instance of that class to the `urllib._urlopener` variable before calling the desired function. For example, applications may want to specify a different `User-Agent` header than `URLopener` defines. This can be accomplished with the following code:

```python
import urllib

class AppURLopener(urllib.FancyURLopener):
    version = "App/1.7"

urllib._urlopener = AppURLopener()
```
urllib. **urlcleanup()**

Clear the cache that may have been built up by previous calls to **urlretrieve()**.
20.5.2. Utility functions

\texttt{urllib.quote(string[, safe])}

Replace special characters in \textit{string} using the \%xx escape. Letters, digits, and the characters '_.-' are never quoted. By default, this function is intended for quoting the path section of the URL. The optional \textit{safe} parameter specifies additional characters that should not be quoted — its default value is '/'.

Example: \texttt{quote('/~connolly/')} yields '/%7econnolly/'.

\texttt{urllib.quote_plus(string[, safe])}

Like \texttt{quote()}, but also replaces spaces by plus signs, as required for quoting HTML form values when building up a query string to go into a URL. Plus signs in the original string are escaped unless they are included in \textit{safe}. It also does not have \textit{safe} default to '/'.

\texttt{urllib.unquote(string)}

Replace \%xx escapes by their single-character equivalent.

Example: \texttt{unquote('/%7Econnolly/')} yields '/~connolly/'.

\texttt{urllib.unquote_plus(string)}

Like \texttt{unquote()}, but also replaces plus signs by spaces, as required for unquoting HTML form values.

\texttt{urllib.urlencode(query[, doseq])}

Convert a mapping object or a sequence of two-element tuples to a “percent-encoded” string, suitable to pass to \texttt{urlopen()} above as the optional \textit{data} argument. This is useful to pass a dictionary of form fields to a \texttt{POST} request. The resulting string is a series of
key=value pairs separated by ' & ' characters, where both key and value are quoted using quote_plus() above. When a sequence of two-element tuples is used as the query argument, the first element of each tuple is a key and the second is a value. The value element in itself can be a sequence and in that case, if the optional parameter doseq is evaluates to True, individual key=value pairs separated by ' & ' are generated for each element of the value sequence for the key. The order of parameters in the encoded string will match the order of parameter tuples in the sequence. The urlparse module provides the functions parse_qs() and parse_qsl() which are used to parse query strings into Python data structures.

urllib.pathname2url(path)
Convert the pathname path from the local syntax for a path to the form used in the path component of a URL. This does not produce a complete URL. The return value will already be quoted using the quote() function.

urllib.url2pathname(path)
Convert the path component path from a percent-encoded URL to the local syntax for a path. This does not accept a complete URL. This function uses unquote() to decode path.

urllib.getproxies()
This helper function returns a dictionary of scheme to proxy server URL mappings. It scans the environment for variables named <scheme>proxy, in case insensitive way, for all operating systems first, and when it cannot find it, looks for proxy information from Mac OSX System Configuration for Mac OS X and Windows Systems Registry for Windows. If both lowercase and uppercase environment variables exist (and disagree), lowercase is preferred.
**Note:** If the environment variable `REQUEST_METHOD` is set, which usually indicates your script is running in a CGI environment, the environment variable `HTTP_PROXY` (uppercase `_PROXY`) will be ignored. This is because that variable can be injected by a client using the “Proxy:” HTTP header. If you need to use an HTTP proxy in a CGI environment, either use `ProxyHandler` explicitly, or make sure the variable name is in lowercase (or at least the `_proxy` suffix).

**Note:** `urllib` also exposes certain utility functions like `splittype`, `splithost` and others parsing URL into various components. But it is recommended to use `urlparse` for parsing URLs rather than using these functions directly. Python 3 does not expose these helper functions from `urllib.parse` module.
**20.5.3. URL Opener objects**

```python
class urllib.URLopener([proxies[, context[, **x509]]])
```

Base class for opening and reading URLs. Unless you need to support opening objects using schemes other than `http:`, `ftp:`, or `file:`, you probably want to use `FancyURLopener`.

By default, the `URLopener` class sends a *User-Agent* header of `urllib/VVV`, where VVV is the `urllib` version number. Applications can define their own *User-Agent* header by subclassing `URLopener` or `FancyURLopener` and setting the class attribute `version` to an appropriate string value in the subclass definition.

The optional `proxies` parameter should be a dictionary mapping scheme names to proxy URLs, where an empty dictionary turns proxies off completely. Its default value is `None`, in which case environmental proxy settings will be used if present, as discussed in the definition of `urlopen()`, above.

The `context` parameter may be a `ssl.SSLContext` instance. If given, it defines the SSL settings the opener uses to make HTTPS connections.

Additional keyword parameters, collected in `x509`, may be used for authentication of the client when using the `https:` scheme. The keywords `key_file` and `cert_file` are supported to provide an SSL key and certificate; both are needed to support client authentication.

`URLopener` objects will raise an `IOError` exception if the server returns an error code.
open(fullurl[, data])

Open fullurl using the appropriate protocol. This method sets up cache and proxy information, then calls the appropriate open method with its input arguments. If the scheme is not recognized, open_unknown() is called. The data argument has the same meaning as the data argument of urlopen().

open_unknown(fullurl[, data])

Overridable interface to open unknown URL types.

retrive(url[, filename[, reporthook[, data]]])

Retrieves the contents of url and places it in filename. The return value is a tuple consisting of a local filename and either a mimetools.Message object containing the response headers (for remote URLs) or None (for local URLs). The caller must then open and read the contents of filename. If filename is not given and the URL refers to a local file, the input filename is returned. If the URL is non-local and filename is not given, the filename is the output of tempfile.mktemp() with a suffix that matches the suffix of the last path component of the input URL. If reporthook is given, it must be a function accepting three numeric parameters. It will be called after each chunk of data is read from the network. reporthook is ignored for local URLs.

If the url uses the http: scheme identifier, the optional data argument may be given to specify a post request (normally the request type is GET). The data argument must in standard application/x-www-form-urlencoded format; see the urlencode() function below.
**version**

Variable that specifies the user agent of the opener object. To get `urllib` to tell servers that it is a particular user agent, set this in a subclass as a class variable or in the constructor before calling the base constructor.

```python
class urllib.FancyURLopener(...)
```

FancyURLopener subclasses `URLopener` providing default handling for the following HTTP response codes: 301, 302, 303, 307 and 401. For the 30x response codes listed above, the `Location` header is used to fetch the actual URL. For 401 response codes (authentication required), basic HTTP authentication is performed. For the 30x response codes, recursion is bounded by the value of the `maxtries` attribute, which defaults to 10.

For all other response codes, the method `http_error_default()` is called which you can override in subclasses to handle the error appropriately.

**Note:** According to the letter of RFC 2616, 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and `urllib` reproduces this behaviour.

The parameters to the constructor are the same as those for `URLopener`.

**Note:**

When performing basic authentication, a FancyURLopener instance calls its `prompt_user_passwd()` method. The default implementation asks the users for the required information on the controlling terminal. A subclass may
override this method to support more appropriate behavior if needed.

The FancyURLopener class offers one additional method that should be overloaded to provide the appropriate behavior:

```python
def prompt_user_passwd(host, realm):
    return (user, password)
```

Return information needed to authenticate the user at the given host in the specified security realm. The return value should be a tuple, (user, password), which can be used for basic authentication.

The implementation prompts for this information on the terminal; an application should override this method to use an appropriate interaction model in the local environment.

```python
exception urllib.ContentTooShortError(msg[, content])
```

This exception is raised when the `urlretrieve()` function detects that the amount of the downloaded data is less than the expected amount (given by the `Content-Length` header). The `content` attribute stores the downloaded (and supposedly truncated) data.

*New in version 2.5.*
20.5.4. urllib Restrictions

- Currently, only the following protocols are supported: HTTP, (versions 0.9 and 1.0), FTP, and local files.

- The caching feature of `urlretrieve()` has been disabled until I find the time to hack proper processing of Expiration time headers.

- There should be a function to query whether a particular URL is in the cache.

- For backward compatibility, if a URL appears to point to a local file but the file can’t be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.

- The `urlopen()` and `urlretrieve()` functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive Web client using these functions without using threads.

- The data returned by `urlopen()` or `urlretrieve()` is the raw data returned by the server. This may be binary data (such as an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the `Content-Type` header. If the returned data is HTML, you can use the module `htmllib` to parse it.

- The code handling the FTP protocol cannot differentiate between a file and a directory. This can lead to unexpected behavior when attempting to read a URL that points to a file that is not accessible. If the URL ends in a `/`, it is assumed to refer to a directory and will be handled accordingly. But if an attempt to
read a file leads to a 550 error (meaning the URL cannot be found or is not accessible, often for permission reasons), then the path is treated as a directory in order to handle the case when a directory is specified by a URL but the trailing / has been left off. This can cause misleading results when you try to fetch a file whose read permissions make it inaccessible; the FTP code will try to read it, fail with a 550 error, and then perform a directory listing for the unreadable file. If fine-grained control is needed, consider using the ftplib module, subclassing FancyURLopener, or changing _urloper to meet your needs.

- This module does not support the use of proxies which require authentication. This may be implemented in the future.

- Although the urllib module contains (undocumented) routines to parse and unparse URL strings, the recommended interface for URL manipulation is in module urlparse.
20.5.5. Examples

Here is an example session that uses the **GET** method to retrieve a URL containing parameters:

```python
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.musical.com/cgi-bin/query?%s")
>>> print f.read()
```

The following example uses the **POST** method instead:

```python
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.musical.com/cgi-bin/query", params)
>>> print f.read()
```

The following example uses an explicitly specified HTTP proxy, overriding environment settings:

```python
>>> import urllib
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.FancyURLopener(proxies)
>>> f = opener.open("http://www.python.org")
>>> f.read()
```

The following example uses no proxies at all, overriding environment settings:

```python
>>> import urllib
>>> opener = urllib.FancyURLopener({})
>>> f = opener.open("http://www.python.org/")
>>> f.read()
```
20.6. \texttt{urllib2} — extensible library for opening URLs

\textbf{Note}: The \texttt{urllib2} module has been split across several modules in Python 3 named \texttt{urllib.request} and \texttt{urllib.error}. The \texttt{2to3} tool will automatically adapt imports when converting your sources to Python 3.

The \texttt{urllib2} module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.

\textbf{See also}: The Requests package is recommended for a higher-level HTTP client interface.

The \texttt{urllib2} module defines the following functions:

\texttt{urllib2.urlopen(url[, data[, timeout[, cafile[, capath[, cadefault[, context]]]]]])}

Open the URL \texttt{url}, which can be either a string or a \texttt{Request} object.

\texttt{data} may be a string specifying additional data to send to the server, or \texttt{None} if no such data is needed. Currently HTTP requests are the only ones that use \texttt{data}; the HTTP request will be a POST instead of a GET when the \texttt{data} parameter is provided. \texttt{data} should be a buffer in the standard \texttt{application/x-www-form-urlencoded} format. The \texttt{urllib.urlencode()} function takes a mapping or sequence of 2-tuples and returns a string in this format. \texttt{urllib2} module sends HTTP/1.1 requests with \texttt{Connection:close} header included.
The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). This actually only works for HTTP, HTTPS and FTP connections.

If *context* is specified, it must be a **ssl.SSLContext** instance describing the various SSL options. See **HTTPSConnection** for more details.

The optional *cafile* and *capath* parameters specify a set of trusted CA certificates for HTTPS requests. *cafile* should point to a single file containing a bundle of CA certificates, whereas *capath* should point to a directory of hashed certificate files. More information can be found in **ssl.SSLContext.load_verify_locations()**.

The *cadefault* parameter is ignored.

This function returns a file-like object with three additional methods:

- **geturl()** — return the URL of the resource retrieved, commonly used to determine if a redirect was followed
- **info()** — return the meta-information of the page, such as headers, in the form of an **mimetypes.Message** instance (see **Quick Reference to HTTP Headers**)
- **getcode()** — return the HTTP status code of the response.

Raises **URLError** on errors.

Note that **None** may be returned if no handler handles the request (though the default installed global **OpenerDirector** uses **UnknownHandler** to ensure this never happens).

In addition, if proxy settings are detected (for example, when a *http_proxy* environment variable like **http_proxy** is set),
**ProxyHandler** is default installed and makes sure the requests are handled through the proxy.

*Changed in version 2.6: timeout was added.*

*Changed in version 2.7.9: cafile, capath, cadefault, and context were added.*

**urllib2.install_opener(opener)**

Install an **OpenerDirector** instance as the default global opener. Installing an opener is only necessary if you want urlopen to use that opener; otherwise, simply call **OpenerDirector.open()** instead of **urlopen()**. The code does not check for a real **OpenerDirector**, and any class with the appropriate interface will work.

**urllib2.build_opener([handler, ...])**

Return an **OpenerDirector** instance, which chains the handlers in the order given. **handlers** can be either instances of **BaseHandler**, or subclasses of **BaseHandler** (in which case it must be possible to call the constructor without any parameters). Instances of the following classes will be in front of the **handlers**, unless the **handlers** contain them, instances of them or subclasses of them: **ProxyHandler** (if proxy settings are detected), **UnknownHandler**, **HTTPHandler**, **HTTPDefaultErrorHandler**, **HTTPRedirectHandler**, **FTPHandler**, **FileHandler**, **HTTPRequestProcessor**.

If the Python installation has SSL support (i.e., if the **ssl** module can be imported), **HTTPSHandler** will also be added.

Beginning in Python 2.3, a **BaseHandler** subclass may also change its **handler_order** attribute to modify its position in the handlers list.

The following exceptions are raised as appropriate:
exception urllib2.URLError
The handlers raise this exception (or derived exceptions) when they run into a problem. It is a subclass of IOError.

reason
The reason for this error. It can be a message string or another exception instance (socket.error for remote URLs, OSError for local URLs).

exception urllib2.HTTPError
Though being an exception (a subclass of URLError), an HTTPError can also function as a non-exceptional file-like return value (the same thing that urlopen() returns). This is useful when handling exotic HTTP errors, such as requests for authentication.

code
An HTTP status code as defined in RFC 2616. This numeric value corresponds to a value found in the dictionary of codes as found in BaseHTTPServer.BaseHTTPRequestHandler.responses.

reason
The reason for this error. It can be a message string or another exception instance.

The following classes are provided:

class urllib2.Request(url[, data][, headers][, origin_req_host][, unverifiable])
This class is an abstraction of a URL request.

url should be a string containing a valid URL.

data may be a string specifying additional data to send to the server, or None if no such data is needed. Currently HTTP
requests are the only ones that use data; the HTTP request will be a POST instead of a GET when the data parameter is provided. data should be a buffer in the standard application/x-www-form-urlencoded format. The urllib.urlencode() function takes a mapping or sequence of 2-tuples and returns a string in this format.

headers should be a dictionary, and will be treated as if add_header() was called with each key and value as arguments. This is often used to “spoof” the User-Agent header value, which is used by a browser to identify itself – some HTTP servers only allow requests coming from common browsers as opposed to scripts. For example, Mozilla Firefox may identify itself as "Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11", while urllib2's default user agent string is "Python-urllib/2.6" (on Python 2.6).

The final two arguments are only of interest for correct handling of third-party HTTP cookies:

origin_req_host should be the request-host of the origin transaction, as defined by RFC 2965. It defaults to cookielib.request_host(self). This is the host name or IP address of the original request that was initiated by the user. For example, if the request is for an image in an HTML document, this should be the request-host of the request for the page containing the image.

unverifiable should indicate whether the request is unverifiable, as defined by RFC 2965. It defaults to False. An unverifiable request is one whose URL the user did not have the option to approve. For example, if the request is for an image in an HTML document, and the user had no option to approve the automatic fetching of the image, this should be true.
**class urllib2.**OpenerDirector

The **OpenerDirector** class opens URLs via **BaseHandler**s chained together. It manages the chaining of handlers, and recovery from errors.

**class urllib2.**BaseHandler

This is the base class for all registered handlers — and handles only the simple mechanics of registration.

**class urllib2.**HTTPDefaultErrorHandler

A class which defines a default handler for HTTP error responses; all responses are turned into **HTTPError** exceptions.

**class urllib2.**HTTPRedirectHandler

A class to handle redirections.

**class urllib2.**HTTPCookieProcessor([cookiejar])

A class to handle HTTP Cookies.

**class urllib2.**ProxyHandler([proxies])

Cause requests to go through a proxy. If **proxies** is given, it must be a dictionary mapping protocol names to URLs of proxies. The default is to read the list of proxies from the environment variables `<protocol>_proxy`. If no proxy environment variables are set, then in a Windows environment proxy settings are obtained from the registry’s Internet Settings section, and in a Mac OS X environment proxy information is retrieved from the OS X System Configuration Framework.

To disable autodetected proxy pass an empty dictionary.

**Note:** `HTTP_PROXY` will be ignored if a variable `REQUEST_METHOD` is set; see the documentation on `getproxies()`.
class urllib2.HTTPPasswordMgr
    Keep a database of \((\text{realm}, \text{uri}) \rightarrow (\text{user}, \text{password})\) mappings.

class urllib2.HTTPPasswordMgrWithDefaultRealm
    Keep a database of \((\text{realm}, \text{uri}) \rightarrow (\text{user}, \text{password})\) mappings. A realm of \text{None} is considered a catch-all realm, which is searched if no other realm fits.

class urllib2.AbstractBasicAuthHandler([\text{password\_mgr}])
    This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. \text{password\_mgr}, if given, should be something that is compatible with \text{HTTPPasswordMgr}; refer to section \text{HTTPPasswordMgr Objects} for information on the interface that must be supported.

class urllib2.HTTPBasicAuthHandler([\text{password\_mgr}])
    Handle authentication with the remote host. \text{password\_mgr}, if given, should be something that is compatible with \text{HTTPPasswordMgr}; refer to section \text{HTTPPasswordMgr Objects} for information on the interface that must be supported.

class urllib2.ProxyBasicAuthHandler([\text{password\_mgr}])
    Handle authentication with the proxy. \text{password\_mgr}, if given, should be something that is compatible with \text{HTTPPasswordMgr}; refer to section \text{HTTPPasswordMgr Objects} for information on the interface that must be supported.

class urllib2.AbstractDigestAuthHandler([\text{password\_mgr}])
    This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. \text{password\_mgr}, if given, should be something that is compatible with \text{HTTPPasswordMgr}; refer to
section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib2.HTTPDigestAuthHandler([password_mgr])
Handle authentication with the remote host. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib2.ProxyDigestAuthHandler([password_mgr])
Handle authentication with the proxy. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib2.HTTPHandler
A class to handle opening of HTTP URLs.

class urllib2.HTTPSHandler([debuglevel[, context]])
A class to handle opening of HTTPS URLs. context has the same meaning as for http://http.client.HTTPSConnection.

   Changed in version 2.7.9: context added.

class urllib2.FileHandler
Open local files.

class urllib2.FTPHandler
Open FTP URLs.

class urllib2.CacheFTPHandler
Open FTP URLs, keeping a cache of open FTP connections to minimize delays.

class urllib2.UnknownHandler
A catch-all class to handle unknown URLs.

class urllib2.HTTPErrorProcessor
    Process HTTP error responses.
20.6.1. Request Objects

The following methods describe all of Request's public interface, and so all must be overridden in subclasses.

Request.**add_data**(data)

Set the Request data to data. This is ignored by all handlers except HTTP handlers — and there it should be a byte string, and will change the request to be POST rather than GET.

Request.**get_method**( )

Return a string indicating the HTTP request method. This is only meaningful for HTTP requests, and currently always returns 'GET' or 'POST'.

Request.**has_data**( )

Return whether the instance has a non-None data.

Request.**get_data**( )

Return the instance's data.

Request.**add_header**(key, val)

Add another header to the request. Headers are currently ignored by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the key collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header.

Request.**add_unredirected_header**(key, header)

Add a header that will not be added to a redirected request.
New in version 2.4.

**Request.** has_header(*header*)

Return whether the instance has the named header (checks both regular and unredirected).

New in version 2.4.

**Request.** get_full_url()

Return the URL given in the constructor.

**Request.** get_type()

Return the type of the URL — also known as the scheme.

**Request.** get_host()

Return the host to which a connection will be made.

**Request.** get_selector()

Return the selector — the part of the URL that is sent to the server.

**Request.** get_header(*header_name*, *default=None*)

Return the value of the given header. If the header is not present, return the default value.

**Request.** header_items()

Return a list of tuples (header_name, header_value) of the Request headers.

**Request.** set_proxy(*host*, *type*)

Prepare the request by connecting to a proxy server. The host and type will replace those of the instance, and the instance’s selector will be the original URL given in the constructor.

**Request.** get_origin_req_host()

Return the request-host of the origin transaction, as defined by
RFC 2965. See the documentation for the Request constructor.

Request.is_unverifiable()

Return whether the request is unverifiable, as defined by RFC 2965. See the documentation for the Request constructor.
20.6.2. OpenerDirector Objects

**OpenerDirector** instances have the following methods:

**OpenerDirector.**\texttt{add\_handler(handler)}

\texttt{handler} should be an instance of **BaseHandler**. The following methods are searched, and added to the possible chains (note that HTTP errors are a special case).

- \texttt{protocol\_open} — signal that the handler knows how to open protocol URLs.
- \texttt{http\_error\_type} — signal that the handler knows how to handle HTTP errors with HTTP error code \texttt{type}.
- \texttt{protocol\_error} — signal that the handler knows how to handle errors from (non-http) \texttt{protocol}.
- \texttt{protocol\_request} — signal that the handler knows how to pre-process \texttt{protocol} requests.
- \texttt{protocol\_response} — signal that the handler knows how to post-process \texttt{protocol} responses.

**OpenerDirector.**\texttt{open(url[, data][, timeout])}

Open the given \texttt{url} (which can be a request object or a string), optionally passing the given \texttt{data}. Arguments, return values and exceptions raised are the same as those of \texttt{urlopen()} (which simply calls the \texttt{open()} method on the currently installed global **OpenerDirector**). The optional \texttt{timeout} parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). The timeout feature actually works only for HTTP, HTTPS and FTP connections).

*Changed in version 2.6: timeout was added.*
**OpenerDirector.error**(proto[, arg[, ...]])

Handle an error of the given protocol. This will call the registered error handlers for the given protocol with the given arguments (which are protocol specific). The HTTP protocol is a special case which uses the HTTP response code to determine the specific error handler; refer to the **http_error_*()** methods of the handler classes.

Return values and exceptions raised are the same as those of urlopen().

OpenerDirector objects open URLs in three stages:

The order in which these methods are called within each stage is determined by sorting the handler instances.

1. Every handler with a method named like **protocol_request** has that method called to pre-process the request.

2. Handlers with a method named like **protocol_open** are called to handle the request. This stage ends when a handler either returns a non-**None** value (ie. a response), or raises an exception (usually **URLError**). Exceptions are allowed to propagate.

In fact, the above algorithm is first tried for methods named **default_open()**. If all such methods return **None**, the algorithm is repeated for methods named like **protocol_open**. If all such methods return **None**, the algorithm is repeated for methods named **unknown_open()**.

Note that the implementation of these methods may involve calls of the parent **OpenerDirector** instance’s **open()** and **error()** methods.
3. Every handler with a method named like `protocol_response` has that method called to post-process the response.
20.6.3. BaseHandler Objects

BaseHandler objects provide a couple of methods that are directly useful, and others that are meant to be used by derived classes. These are intended for direct use:

BaseHandler. **add_parent**(director)

Add a director as parent.

BaseHandler. **close**()

Remove any parents.

The following attributes and methods should only be used by classes derived from BaseHandler.

**Note:** The convention has been adopted that subclasses defining protocol_request() or protocol_response() methods are named *Processor; all others are named *Handler.

BaseHandler. **parent**

A valid OpenerDirector, which can be used to open using a different protocol, or handle errors.

BaseHandler. **default_open**(req)

This method is **not** defined in BaseHandler, but subclasses should define it if they want to catch all URLs.

This method, if implemented, will be called by the parent OpenerDirector. It should return a file-like object as described in the return value of the open() of OpenerDirector, or None. It should raise URLError, unless a truly exceptional thing happens (for example, MemoryError should not be mapped to URLError).
This method will be called before any protocol-specific open method.

BaseHandler. **protocol_open**(req)

(“protocol” is to be replaced by the protocol name.)

This method is *not* defined in BaseHandler, but subclasses should define it if they want to handle URLs with the given *protocol*.

This method, if defined, will be called by the parent OpenerDirector. Return values should be the same as for default_open().

BaseHandler. **unknown_open**(req)

This method is *not* defined in BaseHandler, but subclasses should define it if they want to catch all URLs with no specific registered handler to open it.

This method, if implemented, will be called by the parent OpenerDirector. Return values should be the same as for default_open().

BaseHandler. **http_error_default**(req, fp, code, msg, hdrs)

This method is *not* defined in BaseHandler, but subclasses should override it if they intend to provide a catch-all for otherwise unhandled HTTP errors. It will be called automatically by the OpenerDirector getting the error, and should not normally be called in other circumstances.

*req* will be a Request object, *fp* will be a file-like object with the HTTP error body, *code* will be the three-digit code of the error, *msg* will be the user-visible explanation of the code and *hdrs* will be a mapping object with the headers of the error.

Return values and exceptions raised should be the same as
those of `urlopen()`.

`BaseHandler.http_error_nnn(req, fp, code, msg, hdrs)`

`nnn` should be a three-digit HTTP error code. This method is also not defined in `BaseHandler`, but will be called, if it exists, on an instance of a subclass, when an HTTP error with code `nnn` occurs.

Subclasses should override this method to handle specific HTTP errors.

Arguments, return values and exceptions raised should be the same as for `http_error_default()`.

`BaseHandler.protocol_request(req)`

(“protocol” is to be replaced by the protocol name.)

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to pre-process requests of the given `protocol`.

This method, if defined, will be called by the parent `OpenerDirector`. `req` will be a `Request` object. The return value should be a `Request` object.

`BaseHandler.protocol_response(req, response)`

(“protocol” is to be replaced by the protocol name.)

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to post-process responses of the given `protocol`.

This method, if defined, will be called by the parent `OpenerDirector`. `req` will be a `Request` object. `response` will be an object implementing the same interface as the return value of
The return value should implement the same interface as the return value of `urlopen()`.
20.6.4. HTTPRedirectHandler Objects

**Note:** Some HTTP redirections require action from this module's client code. If this is the case, `HTTPError` is raised. See RFC 2616 for details of the precise meanings of the various redirection codes.

HTTPRedirectHandler. `redirect_request(req, fp, code, msg, hdrs, newurl)`

Return a `Request` or `None` in response to a redirect. This is called by the default implementations of the `http_error_30*()` methods when a redirection is received from the server. If a redirection should take place, return a new `Request` to allow `http_error_30*()` to perform the redirect to `newurl`. Otherwise, raise `HTTPError` if no other handler should try to handle this URL, or return `None` if you can’t but another handler might.

**Note:** The default implementation of this method does not strictly follow RFC 2616, which says that 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and the default implementation reproduces this behavior.

HTTPRedirectHandler. `http_error_301(req, fp, code, msg, hdrs)`

Redirect to the `Location:` or `URI:` URL. This method is called by the parent `OpenerDirector` when getting an HTTP ‘moved permanently’ response.

HTTPRedirectHandler. `http_error_302(req, fp, code, msg, hdrs)`

The same as `http_error_301()`, but called for the ‘found’ response.
HTTPRedirectHandler.\texttt{http\_error\_303}(\textit{req, fp, code, msg, hdrs})

The same as \texttt{http\_error\_301()}, but called for the ‘see other’ response.

HTTPRedirectHandler.\texttt{http\_error\_307}(\textit{req, fp, code, msg, hdrs})

The same as \texttt{http\_error\_301()}, but called for the ‘temporary redirect’ response.
20.6.5. HTTPCookieProcessor Objects

New in version 2.4.

HTTPCookieProcessor instances have one attribute:

HTTPCookieProcessor.singleton.cookiejar

The cookielib.CookieJar in which cookies are stored.
20.6.6. ProxyHandler Objects

ProxyHandler. `protocol_open(request)`

(“protocol” is to be replaced by the protocol name.)

The `ProxyHandler` will have a method `protocol_open` for every `protocol` which has a proxy in the `proxies` dictionary given in the constructor. The method will modify requests to go through the proxy, by calling `request.set_proxy()`, and call the next handler in the chain to actually execute the protocol.
20.6.7. HTTPPasswordMgr Objects

These methods are available on HTTPPasswordMgr and HTTPPasswordMgrWithDefaultRealm objects.

HTTPPasswordMgr.\texttt{add\_password}(\texttt{realm}, \texttt{uri}, \texttt{user}, \texttt{passwd})

\textit{uri} can be either a single URI, or a sequence of URIs. \textit{realm}, \textit{user} and \textit{passwd} must be strings. This causes \texttt{(user, passwd)} to be used as authentication tokens when authentication for \textit{realm} and a super-URI of any of the given URIs is given.

HTTPPasswordMgr.\texttt{find\_user\_password}(\texttt{realm}, \texttt{authuri})

Get user/password for given realm and URI, if any. This method will return \texttt{(None, None)} if there is no matching user/password.

For HTTPPasswordMgrWithDefaultRealm objects, the realm \texttt{None} will be searched if the given \textit{realm} has no matching user/password.
20.6.8. AbstractBasicAuthHandler Objects

AbstractBasicAuthHandler. `http_error_auth_reqed(authreq, host, req, headers)`

Handle an authentication request by getting a user/password pair, and re-trying the request. `authreq` should be the name of the header where the information about the realm is included in the request, `host` specifies the URL and path to authenticate for, `req` should be the (failed) `Request` object, and `headers` should be the error headers.

`host` is either an authority (e.g. "python.org") or a URL containing an authority component (e.g. `http://python.org/`). In either case, the authority must not contain a userinfo component (so, "python.org" and "python.org:80" are fine, "joe:password@python.org" is not).
HTTPBasicAuthHandler. `http_error_401(req, fp, code, msg, hdrs)`

Retry the request with authentication information, if available.
ProxyBasicAuthHandler.

ProxyBasicAuthHandler.http_error_407(req, fp, code, msg, hdrs)

Retry the request with authentication information, if available.
20.6.11. AbstractDigestAuthHandler Objects

AbstractDigestAuthHandler. **http_error_auth_reqed** (authreq, host, req, headers)

*authreq* should be the name of the header where the information about the realm is included in the request, *host* should be the host to authenticate to, *req* should be the (failed) Request object, and *headers* should be the error headers.
20.6.12. HTTPDigestAuthHandler Objects

HTTPDigestAuthHandler.**http_error_401**(req, fp, code, msg, hdrs)

Retry the request with authentication information, if available.
20.6.13. ProxyDigestAuthHandler Objects

ProxyDigestAuthHandler.\texttt{http\_error\_407}(req, fp, code, msg, hdrs)

Retry the request with authentication information, if available.
HTTPHandler.http_open(req)
Send an HTTP request, which can be either GET or POST, depending on req.has_data().
Send an HTTPS request, which can be either GET or POST, depending on `req.has_data()`.
20.6.16. FileHandler Objects

FileHandler. file_open(req)

Open the file locally, if there is no host name, or the host name is 'localhost'. Change the protocol to ftp otherwise, and retry opening it using parent.
20.6.17. FTPHandler Objects

FTPHandler. **ftp_open**(req)

Open the FTP file indicated by `req`. The login is always done with empty username and password.
20.6.18. CacheFTPHandler Objects

*CacheFTPHandler* objects are *FTPHandler* objects with the following additional methods:

**CacheFTPHandler. `setTimeout(t)`**
Set timeout of connections to \( t \) seconds.

**CacheFTPHandler. `setMaxConns(m)`**
Set maximum number of cached connections to \( m \).
20.6.19. UnknownHandler Objects

UnknownHandler. **unknown_open()**

Raise a **URL Error** exception.
20.6.20. HTTPErrorProcessor Objects

New in version 2.4.

HTTPErrorProcessor. **http_response()**
Process HTTP error responses.

For 200 error codes, the response object is returned immediately.

For non-200 error codes, this simply passes the job on to the
`protocol_error_code` handler methods, via
`OpenerDirector.error()`.

Eventually, `urllib2.HTTPDefaultErrorHandler` will raise an `HTTPError` if no
other handler handles the error.

HTTPErrorProcessor. **https_response()**
Process HTTPS error responses.

The behavior is same as `http_response()`.
In addition to the examples below, more examples are given in HOWTO Fetch Internet Resources Using urllib2.

This example gets the python.org main page and displays the first 100 bytes of it:

```python
>>> import urllib2
>>> f = urllib2.urlopen('http://www.python.org/)
>>> print f.read(100)
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<xml-stylesheet href="./css/ht2html

Here we are sending a data-stream to the stdin of a CGI and reading the data it returns to us. Note that this example will only work when the Python installation supports SSL.

```python
>>> import urllib2
>>> req = urllib2.Request(url='https://localhost/cgi-bin/test.cgi
... data='This data is passed to stdin of
>>> f = urllib2.urlopen(req)
>>> print f.read()
Got Data: "This data is passed to stdin of the CGI"

The code for the sample CGI used in the above example is:

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print 'Content-type: text-plain\n\nGot Data: "%s"' % data
```

Use of Basic HTTP Authentication:

```
import urllib2
# Create an OpenerDirector with support for Basic HTTP Authenti
auth_handler = urllib2.HTTPBasicAuthHandler()
```
auth_handler.add_password(realm='PDQ Application',
    uri='https://mahler:8092/site-updates
    user='klem',
    passwd='kadidd!ehopper')

opener = urllib2.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib2.install_opener(opener)
urllib2.urlopen('http://www.example.com/login.html')

build_opener() provides many handlers by default, including a
ProxyHandler. By default, ProxyHandler uses the environment
variables named <scheme>_proxy, where <scheme> is the URL
scheme involved. For example, the http_proxy environment variable
is read to obtain the HTTP proxy's URL.

This example replaces the default ProxyHandler with one that uses
programmatically-supplied proxy URLs, and adds proxy
authorization support with ProxyBasicAuthHandler.

proxy_handler = urllib2.ProxyHandler({'http': 'http://www.examp
proxy_auth_handler = urllib2.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'p

opener = urllib2.build_opener(proxy_handler, proxy_auth_handler
# This time, rather than install the OpenerDirector, we use it
opener.open('http://www.example.com/login.html')

Adding HTTP headers:

Use the headers argument to the Request constructor, or:

import urllib2
req = urllib2.Request('http://www.example.com/)
req.add_header('Referer', 'http://www.python.org/)
# Customize the default User-Agent header value:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: ..
r = urllib2.urlopen(req)
**OpenerDirector** automatically adds a *User-Agent* header to every *Request*. To change this:

```python
import urllib2
opener = urllib2.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/
```

Also, remember that a few standard headers (*Content-Length*, *Content-Type* and *Host*) are added when the *Request* is passed to *urlopen()* (or *OpenerDirector.open()*).
20.7. `httplib` — HTTP protocol client

**Note:** The `httplib` module has been renamed to `http.client` in Python 3. The `2to3` tool will automatically adapt imports when converting your sources to Python 3.

**Source code:** `Lib/httplib.py`

This module defines classes which implement the client side of the HTTP and HTTPS protocols. It is normally not used directly — the module `urllib` uses it to handle URLs that use HTTP and HTTPS.

**See also:** The Requests package is recommended for a higher-level HTTP client interface.

**Note:** HTTPS support is only available if the `socket` module was compiled with SSL support.

**Note:** The public interface for this module changed substantially in Python 2.0. The `HTTP` class is retained only for backward compatibility with 1.5.2. It should not be used in new code. Refer to the online docstrings for usage.

The module provides the following classes:

```python
class httpclient.HTTPConnection(host[, port[, strict[, timeout[, source_address]]]])
```

An `HTTPConnection` instance represents one transaction with an HTTP server. It should be instantiated passing it a host and optional port number. If no port number is passed, the port is
extracted from the host string if it has the form `host:port`, else the default HTTP port (80) is used. When true, the optional parameter `strict` (which defaults to a false value) causes `BadStatusLine` to be raised if the status line can’t be parsed as a valid HTTP/1.0 or 1.1 status line. If the optional `timeout` parameter is given, blocking operations (like connection attempts) will timeout after that many seconds (if it is not given, the global default timeout setting is used). The optional `source_address` parameter may be a tuple of a (host, port) to use as the source address the HTTP connection is made from.

For example, the following calls all create instances that connect to the server at the same host and port:

```python
>>> h1 = httplib.HTTPConnection('www.cwi.nl')
>>> h2 = httplib.HTTPConnection('www.cwi.nl:80')
>>> h3 = httplib.HTTPConnection('www.cwi.nl', 80)
>>> h3 = httplib.HTTPConnection('www.cwi.nl', 80, timeout=10)
```

*New in version 2.0.*

*Changed in version 2.6: timeout was added.*

*Changed in version 2.7: source_address was added.*

class `httplib.HTTPSConnection`(*host*, *port*, *key_file*, *cert_file*,
`strict`, *timeout*, *source_address*, *context*)

A subclass of `HTTPConnection` that uses SSL for communication with secure servers. Default port is 443. If `context` is specified, it must be a `ssl.SSLContext` instance describing the various SSL options.

`key_file` and `cert_file` are deprecated, please use `ssl.SSLContext.load_cert_chain()` instead, or let
ssl.create_default_context() select the system’s trusted CA certificates for you.

Please read Security considerations for more information on best practices.

New in version 2.0.

Changed in version 2.6: timeout was added.

Changed in version 2.7: source_address was added.

Changed in version 2.7.9: context was added.

This class now performs all the necessary certificate and hostname checks by default. To revert to the previous, unverified, behavior ssl._create_unverified_context() can be passed to the context parameter.

class httplib.HTTPResponse(sock, debuglevel=0, strict=0)

Class whose instances are returned upon successful connection. Not instantiated directly by user.

New in version 2.0.

class httplib.HTTPMessage

An HTTPMessage instance is used to hold the headers from an HTTP response. It is implemented using the mimetools.Message class and provides utility functions to deal with HTTP Headers. It is not directly instantiated by the users.

The following exceptions are raised as appropriate:

exception httplib.HTTPException

The base class of the other exceptions in this module. It is a subclass of Exception.
New in version 2.0.

exception `http://NotConnected`
    A subclass of `HTTPException`.

New in version 2.0.

exception `http://InvalidURL`
    A subclass of `HTTPException`, raised if a port is given and is either non-numeric or empty.

New in version 2.3.

exception `http://UnknownProtocol`
    A subclass of `HTTPException`.

New in version 2.0.

exception `http://UnknownTransferEncoding`
    A subclass of `HTTPException`.

New in version 2.0.

exception `http://UnimplementedFileMode`
    A subclass of `HTTPException`.

New in version 2.0.

exception `http://IncompleteRead`
    A subclass of `HTTPException`.

New in version 2.0.

exception `http://ImproperConnectionState`
    A subclass of `HTTPException`. 
New in version 2.0.

`exception` `httplib.CannotSendRequest`  
A subclass of `ImproperConnectionState`.

New in version 2.0.

`exception` `httplib.CannotSendHeader`  
A subclass of `ImproperConnectionState`.

New in version 2.0.

`exception` `httplib.ResponseNotReady`  
A subclass of `ImproperConnectionState`.

New in version 2.0.

`exception` `httplib.BadStatusLine`  
A subclass of `HTTPException`. Raised if a server responds with a HTTP status code that we don’t understand.

New in version 2.0.

The constants defined in this module are:

`httplib.HTTP_PORT`  
The default port for the HTTP protocol (always 80).

`httplib.HTTPS_PORT`  
The default port for the HTTPS protocol (always 443).

and also the following constants for integer status codes:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTINUE</td>
<td>100</td>
<td>HTTP/1.1, RFC 2616, Section 10.1.1</td>
</tr>
<tr>
<td>Status Code</td>
<td>Status Description</td>
<td>Protocol Version</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>101</td>
<td>SWITCHING_PROTOCOLS</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>102</td>
<td>PROCESSING</td>
<td>WEBDAV</td>
</tr>
<tr>
<td>200</td>
<td>OK</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>201</td>
<td>CREATED</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>202</td>
<td>ACCEPTED</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>203</td>
<td>NON_AUTHORITATIVE_INFORMATION</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>204</td>
<td>NO_CONTENT</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>205</td>
<td>RESET_CONTENT</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>206</td>
<td>PARTIAL_CONTENT</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>207</td>
<td>MULTI_STATUS</td>
<td>WEBDAV</td>
</tr>
<tr>
<td>226</td>
<td>IM_USED</td>
<td>Delta encoding in HTTP, RFC 3229</td>
</tr>
<tr>
<td>300</td>
<td>MULTIPLE_CHOICES</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>301</td>
<td>MOVED_PERMANENTLY</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>302</td>
<td>FOUND</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>303</td>
<td>SEE_OTHER</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>304</td>
<td>NOT_MODIFIED</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>305</td>
<td>USE_PROXY</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>Temporary Status Code</td>
<td>Numeric Code</td>
<td>HTTP/1.1, RFC 2616, Section</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>TEMPORARY_REDIRECT</td>
<td>307</td>
<td>10.3.8</td>
</tr>
<tr>
<td>BAD_REQUEST</td>
<td>400</td>
<td>10.4.1</td>
</tr>
<tr>
<td>UNAUTHORIZED</td>
<td>401</td>
<td>10.4.2</td>
</tr>
<tr>
<td>PAYMENT_REQUIRED</td>
<td>402</td>
<td>10.4.3</td>
</tr>
<tr>
<td>FORBIDDEN</td>
<td>403</td>
<td>10.4.4</td>
</tr>
<tr>
<td>NOT_FOUND</td>
<td>404</td>
<td>10.4.5</td>
</tr>
<tr>
<td>METHOD_NOT_ALLOWED</td>
<td>405</td>
<td>10.4.6</td>
</tr>
<tr>
<td>NOT_ACCEPTABLE</td>
<td>406</td>
<td>10.4.7</td>
</tr>
<tr>
<td>PROXY_AUTHENTICATION_REQUIRED</td>
<td>407</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.8</td>
</tr>
<tr>
<td>REQUEST_TIMEOUT</td>
<td>408</td>
<td>10.4.9</td>
</tr>
<tr>
<td>CONFLICT</td>
<td>409</td>
<td>10.4.10</td>
</tr>
<tr>
<td>GONE</td>
<td>410</td>
<td>10.4.11</td>
</tr>
<tr>
<td>LENGTH_REQUIRED</td>
<td>411</td>
<td>10.4.12</td>
</tr>
<tr>
<td>PRECONDITION_FAILED</td>
<td>412</td>
<td>10.4.13</td>
</tr>
<tr>
<td>REQUEST_ENTITY_TOO_LARGE</td>
<td>413</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.14</td>
</tr>
<tr>
<td>REQUEST_URI_TOO_LONG</td>
<td>414</td>
<td>10.4.15</td>
</tr>
<tr>
<td>UNSUPPORTED_MEDIA_TYPE</td>
<td>415</td>
<td>10.4.16</td>
</tr>
<tr>
<td>REQUESTED_RANGE_NOT_SATISFIABLE</td>
<td>416</td>
<td>HTTP/1.1, RFC 2616,</td>
</tr>
<tr>
<td>Status Code</td>
<td>HTTP Version</td>
<td>RFC</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td>-----</td>
</tr>
<tr>
<td>417</td>
<td>HTTP/1.1</td>
<td>RFC 2616</td>
</tr>
<tr>
<td>422</td>
<td>WEBDAV</td>
<td>RFC 2518</td>
</tr>
<tr>
<td>423</td>
<td>WEBDAV</td>
<td>RFC 2518</td>
</tr>
<tr>
<td>424</td>
<td>WEBDAV</td>
<td>RFC 2518</td>
</tr>
<tr>
<td>426</td>
<td>HTTP Upgrade</td>
<td>RFC 2817</td>
</tr>
<tr>
<td>500</td>
<td>HTTP/1.1</td>
<td>RFC 2616</td>
</tr>
<tr>
<td>501</td>
<td>HTTP/1.1</td>
<td>RFC 2616</td>
</tr>
<tr>
<td>502</td>
<td>HTTP/1.1</td>
<td>RFC 2616</td>
</tr>
<tr>
<td>503</td>
<td>HTTP/1.1</td>
<td>RFC 2616</td>
</tr>
<tr>
<td>504</td>
<td>HTTP/1.1</td>
<td>RFC 2616</td>
</tr>
<tr>
<td>505</td>
<td>HTTP/1.1</td>
<td>RFC 2616</td>
</tr>
<tr>
<td>507</td>
<td>WEBDAV</td>
<td>RFC 2518</td>
</tr>
<tr>
<td>510</td>
<td>An HTTP Extension Framework</td>
<td>RFC 2774</td>
</tr>
</tbody>
</table>

**httplib.responses**

This dictionary maps the HTTP 1.1 status codes to the W3C names.

Example: `httplib.responses[httplib.NOT_FOUND]` is 'Not Found'.

httplib.responses

This dictionary maps the HTTP 1.1 status codes to the W3C names.

Example: `httplib.responses[httplib.NOT_FOUND]` is 'Not Found'.
New in version 2.5.
20.7.1. HTTPConnection Objects

HTTPConnection instances have the following methods:

HTTPConnection. request( method, url[, body[, headers]])

This will send a request to the server using the HTTP request method method and the selector url. If the body argument is present, it should be a string of data to send after the headers are finished. Alternatively, it may be an open file object, in which case the contents of the file is sent; this file object should support fileno() and read() methods. The headers argument should be a mapping of extra HTTP headers to send with the request.

If one is not provided in headers, a Content-Length header is added automatically for all methods if the length of the body can be determined, either from the length of the str representation, or from the reported size of the file on disk. If body is None the header is not set except for methods that expect a body (PUT, POST, and PATCH) in which case it is set to 0.

Changed in version 2.6: body can be a file object.

HTTPConnection. getresponse()

Should be called after a request is sent to get the response from the server. Returns an HTTPResponse instance.

**Note:** Note that you must have read the whole response before you can send a new request to the server.

HTTPConnection. set_debuglevel(level)

Set the debugging level (the amount of debugging output printed). The default debug level is 0, meaning no debugging
output is printed.

**HTTPConnection.set_tunnel(host, port=None, headers=None)**

Set the host and the port for HTTP Connect Tunnelling. Normally used when it is required to do HTTPS Connection through a proxy server.

The headers argument should be a mapping of extra HTTP headers to send with the CONNECT request.

*New in version 2.7.*

**HTTPConnection.connect()**

Connect to the server specified when the object was created.

**HTTPConnection.close()**

Close the connection to the server.

As an alternative to using the `request()` method described above, you can also send your request step by step, by using the four functions below.

**HTTPConnection.putrequest(request, selector[, skip_host[, skip_accept_encoding]])**

This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the `request` string, the `selector` string, and the HTTP version (`HTTP/1.1`). To disable automatic sending of `Host:` or `Accept-Encoding:` headers (for example to accept additional content encodings), specify `skip_host` or `skip_accept_encoding` with non-False values.

*Changed in version 2.4: skip_accept_encoding argument added.*

**HTTPConnection.putheader(header, argument[, ...])**

Send an **RFC 822**-style header to the server. It sends a line to
the server consisting of the header, a colon and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

`HTTPConnection.endheaders(message_body=None)`

Send a blank line to the server, signalling the end of the headers. The optional `message_body` argument can be used to pass a message body associated with the request. The message body will be sent in the same packet as the message headers if it is string, otherwise it is sent in a separate packet.

*Changed in version 2.7: message_body was added.*

`HTTPConnection.send(data)`

Send data to the server. This should be used directly only after the `endheaders()` method has been called and before `getresponse()` is called.
20.7.2. HTTPResponse Objects

HTTPResponse instances have the following methods and attributes:

HTTPResponse. `read([amt])`
Reads and returns the response body, or up to the next `amt` bytes.

HTTPResponse. `getheader(name[, default])`
Get the contents of the header `name`, or `default` if there is no matching header.

HTTPResponse. `getheaders()`
Return a list of (header, value) tuples.

*New in version 2.4.*

HTTPResponse. `fileno()`
Returns the `fileno` of the underlying socket.

HTTPResponse. `msg`
A `mimetypes.Message` instance containing the response headers.

HTTPResponse. `version`
HTTP protocol version used by server. 10 for HTTP/1.0, 11 for HTTP/1.1.

HTTPResponse. `status`
Status code returned by server.

HTTPResponse. `reason`
Reason phrase returned by server.
20.7.3. Examples

Here is an example session that uses the `GET` method:

```python
>>> import httplib
>>> conn = httplib.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print r1.status, r1.reason
200 OK
>>> data1 = r1.read()
>>> conn.request("GET", "/")
>>> r2 = conn.getresponse()
>>> print r2.status, r2.reason
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

Here is an example session that uses the `HEAD` method. Note that the `HEAD` method never returns any data.

```python
>>> import httplib
>>> conn = httplib.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> res = conn.getresponse()
>>> print res.status, res.reason
200 OK
>>> data = res.read()
>>> print len(data)
0
>>> data == ''
True
```

Here is an example session that shows how to `POST` requests:

```python
>>> import httplib, urllib
>>> params = urllib.urlencode({'@number': 12524, '@type': 'issue'}
>>> headers = {'Content-type': 'application/x-www-form-urlencoded',
...            'Accept': 'text/plain'}
>>> conn = httplib.HTTPConnection("bugs.python.org")
```
Client side **HTTP PUT** requests are very similar to **POST** requests. The difference lies only the server side where HTTP server will allow resources to be created via **PUT** request. Here is an example session that shows how to do **PUT** request using **httplib**:

```python
>>> # This creates an HTTP message
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/foobar
...  
>>> import httplib
>>> BODY = "***filecontents***"
>>> conn = httplib.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print response.status, response.reason
200, OK
```
20.8. ftplib — FTP protocol client

Source code: Lib/ftplib.py

This module defines the class FTP and a few related items. The FTP class implements the client side of the FTP protocol. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other FTP servers. It is also used by the module urllib to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see Internet RFC 959.

Here's a sample session using the ftplib module:

```python
>>> from ftplib import FTP
>>> ftp = FTP('ftp.debian.org')  # connect to host, default
>>> ftp.login()  # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')  # change into "debian" directory
>>> ftp.retrlines('LIST')  # list directory contents
-rw-rw-r--  1 1176  1176    1063 Jun 15 10:18 README
... 
drwxr-sr-x  5 1176  1176    4096 Dec 19 2000 pool
drwxr-sr-x  4 1176  1176    4096 Nov 17 2008 project
drwxr-xr-x  3 1176  1176    4096 Oct 10 2012 tools
'226 Directory send OK.'
>>> ftp.retrbinary('RETR README', open('README', 'wb').write)
'226 Transfer complete.'
>>> ftp.quit()
```

The module defines the following items:

```python
class ftplib.FTP([host[, user[, passwd[, acct[, timeout]]]]])
```

Return a new instance of the FTP class. When host is given, the method call connect(host) is made. When user is given, additionally the method call login(user, passwd, acct) is made.
(where passwd and acct default to the empty string when not given). The optional timeout parameter specifies a timeout in seconds for blocking operations like the connection attempt (if is not specified, the global default timeout setting will be used).

*Changed in version 2.6: timeout was added.*

class ftplib.FTP_TLS([host[, user[, passwd[, acct[, keyfile[, certfile[, context[, timeout]]]]]]]])

A FTP subclass which adds TLS support to FTP as described in RFC 4217. Connect as usual to port 21 implicitly securing the FTP control connection before authenticating. Securing the data connection requires the user to explicitly ask for it by calling the prot_p() method. context is a ssl.SSLContext object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read Security considerations for best practices.

keyfile and certfile are a legacy alternative to context – they can point to PEM-formatted private key and certificate chain files (respectively) for the SSL connection.

*New in version 2.7.*

*Changed in version 2.7.10: The context parameter was added.*

Here’s a sample session using the FTP_TLS class:

```python
>>> from ftplib import FTP_TLS
>>> ftps = FTP_TLS('ftp.python.org')
>>> ftps.login()   # login anonymously before securing control channel
>>> ftps.prot_p()  # switch to secure data connection
>>> ftps.retrlines('LIST') # list directory content securely
total 9
drwxr-xr-x  8 root wheel    1024 Jan 3 1994 .
drwxr-xr-x  8 root wheel    1024 Jan 3 1994..
drwxr-xr-x  2 root wheel    1024 Jan 3 1994 bin
```
exception ftplib.error_reply
Exception raised when an unexpected reply is received from the server.

exception ftplib.error_temp
Exception raised when an error code signifying a temporary error (response codes in the range 400–499) is received.

exception ftplib.error_perm
Exception raised when an error code signifying a permanent error (response codes in the range 500–599) is received.

exception ftplib.error_proto
Exception raised when a reply is received from the server that does not fit the response specifications of the File Transfer Protocol, i.e. begin with a digit in the range 1–5.

ftplib.all_errors
The set of all exceptions (as a tuple) that methods of FTP instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed above as well as socket.error and IOError.

See also:
Module netrc
Parser for the .netrc file format. The file .netrc is typically used by FTP clients to load user authentication information before prompting the user.

The file Tools/scripts/ftpmirror.py in the Python source distribution is a script that can mirror FTP sites, or portions thereof, using the ftplib module. It can be used as an extended example that applies this module.
20.8.1. FTP Objects

Several methods are available in two flavors: one for handling text files and another for binary files. These are named for the command which is used followed by `lines` for the text version or `binary` for the binary version.

FTP instances have the following methods:

FTP. `set_debuglevel(level)`
Set the instance’s debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

FTP. `connect(host[, port[, timeout]])`
Connect to the given host and port. The default port number is 21, as specified by the FTP protocol specification. It is rarely needed to specify a different port number. This function should be called only once for each instance; it should not be called at all if a host was given when the instance was created. All other methods can only be used after a connection has been made.

The optional `timeout` parameter specifies a timeout in seconds for the connection attempt. If no `timeout` is passed, the global default timeout setting will be used.

*Changed in version 2.6: timeout was added.*

FTP. `getwelcome()`
Return the welcome message sent by the server in reply to the
initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

FTP. **login([user[, passwd[, acct]]])**

Log in as the given *user*. The *passwd* and *acct* parameters are optional and default to the empty string. If no *user* is specified, it defaults to 'anonymous'. If *user* is 'anonymous', the default *passwd* is 'anonymous@'. This function should be called only once for each instance, after a connection has been established; it should not be called at all if a host and user were given when the instance was created. Most FTP commands are only allowed after the client has logged in. The *acct* parameter supplies “accounting information”; few systems implement this.

FTP. **abort()**

Abort a file transfer that is in progress. Using this does not always work, but it’s worth a try.

FTP. **sendcmd(command)**

Send a simple command string to the server and return the response string.

FTP. **voidcmd(command)**

Send a simple command string to the server and handle the response. Return nothing if a response code corresponding to success (codes in the range 200–299) is received. Raise `error_reply` otherwise.

FTP. **retrbinary(command, callback[, maxblocksize[, rest]])**

Retrieve a file in binary transfer mode. *command* should be an appropriate RETR command: 'RETR filename'. The *callback* function is called for each block of data received, with a single string argument giving the data block. The optional *maxblocksize* argument specifies the maximum chunk size to read on the low-
level socket object created to do the actual transfer (which will also be the largest size of the data blocks passed to callback). A reasonable default is chosen. rest means the same thing as in the transfercmd() method.

FTP. `retrlines(command[, callback])`
Retrieve a file or directory listing in ASCII transfer mode. command should be an appropriate RETR command (see retrbinary()) or a command such as LIST, NLST or MLSD (usually just the string 'LIST'). LIST retrieves a list of files and information about those files. NLST retrieves a list of file names. On some servers, MLSD retrieves a machine readable list of files and information about those files. The callback function is called for each line with a string argument containing the line with the trailing CRLF stripped. The default callback prints the line to sys.stdout.

FTP. `set_pasv(boolean)`
Enable “passive” mode if boolean is true, other disable passive mode. (In Python 2.0 and before, passive mode was off by default; in Python 2.1 and later, it is on by default.)

FTP. `storbinary(command, file[, blocksize, callback, rest])`
Store a file in binary transfer mode. command should be an appropriate STOR command: "STOR filename". file is an open file object which is read until EOF using its read() method in blocks of size blocksize to provide the data to be stored. The blocksize argument defaults to 8192. callback is an optional single parameter callable that is called on each block of data after it is sent. rest means the same thing as in the transfercmd() method.

Changed in version 2.1: default for blocksize added.


*Changed in version 2.6:* callback parameter added.

*Changed in version 2.7:* rest parameter added.

**FTP. storlines(command, file[, callback])**

Store a file in ASCII transfer mode. command should be an appropriate STOR command (see storbinary()). Lines are read until EOF from the open file object file using its readline() method to provide the data to be stored. callback is an optional single parameter callable that is called on each line after it is sent.

*Changed in version 2.6:* callback parameter added.

**FTP. transfercmd(cmd[, rest])**

Initiate a transfer over the data connection. If the transfer is active, send an EPRT or PORT command and the transfer command specified by cmd, and accept the connection. If the server is passive, send an EPSV or PASV command, connect to it, and start the transfer command. Either way, return the socket for the connection.

If optional rest is given, a REST command is sent to the server, passing rest as an argument. rest is usually a byte offset into the requested file, telling the server to restart sending the file's bytes at the requested offset, skipping over the initial bytes. Note however that RFC 959 requires only that rest be a string containing characters in the printable range from ASCII code 33 to ASCII code 126. The transfercmd() method, therefore, converts rest to a string, but no check is performed on the string's contents. If the server does not recognize the REST command, an error_reply exception will be raised. If this happens, simply call transfercmd() without a rest argument.
FTP. `ntransfercmd(cmd[, rest])`
Like `transfercmd()`, but returns a tuple of the data connection and the expected size of the data. If the expected size could not be computed, `None` will be returned as the expected size. `cmd` and `rest` means the same thing as in `transfercmd()`.

FTP. `nlst(argument[, ...])`
Return a list of file names as returned by the `NLST` command. The optional `argument` is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `NLST` command.

FTP. `dir(argument[, ...])`
Produce a directory listing as returned by the `LIST` command, printing it to standard output. The optional `argument` is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `LIST` command. If the last argument is a function, it is used as a `callback` function as for `retrlines()`; the default prints to `sys.stdout`. This method returns `None`.

FTP. `rename(fromname, toname)`
Rename file `fromname` on the server to `toname`.

FTP. `delete(filename)`
Remove the file named `filename` from the server. If successful, returns the text of the response, otherwise raises `error_perm` on permission errors or `error_reply` on other errors.

FTP. `cwd(pathname)`
Set the current directory on the server.

FTP. `mkd(pathname)`
Create a new directory on the server.

FTP. `pwd()`
Return the pathname of the current directory on the server.

FTP. `rmd(dirname)`
Remove the directory named `dirname` on the server.

FTP. `size(filename)`
Request the size of the file named `filename` on the server. On success, the size of the file is returned as an integer, otherwise `None` is returned. Note that the `SIZE` command is not standardized, but is supported by many common server implementations.

FTP. `quit()`
Send a `QUIT` command to the server and close the connection. This is the “polite” way to close a connection, but it may raise an exception if the server responds with an error to the `QUIT` command. This implies a call to the `close()` method which renders the FTP instance useless for subsequent calls (see below).

FTP. `close()`
Close the connection unilaterally. This should not be applied to an already closed connection such as after a successful call to `quit()`. After this call the FTP instance should not be used any more (after a call to `close()` or `quit()` you cannot reopen the connection by issuing another `login()` method).
20.8.2. FTP_TLS Objects

**FTP_TLS** class inherits from **FTP**, defining these additional objects:

**FTP_TLS.** *ssl_version*

The SSL version to use (defaults to `ssl.PROTOCOL_SSLv23`).

**FTP_TLS.** *auth()*

Set up secure control connection by using TLS or SSL, depending on what specified in `ssl_version()` attribute.

**FTP_TLS.** *prot_p()*

Set up secure data connection.

**FTP_TLS.** *prot_c()*

Set up clear text data connection.
This module defines a class, **POP3**, which encapsulates a connection to a POP3 server and implements the protocol as defined in RFC 1725. The **POP3** class supports both the minimal and optional command sets. Additionally, this module provides a class **POP3_SSL**, which provides support for connecting to POP3 servers that use SSL as an underlying protocol layer.

Note that POP3, though widely supported, is obsolescent. The implementation quality of POP3 servers varies widely, and too many are quite poor. If your mailserver supports IMAP, you would be better off using the `imaplib.IMAP4` class, as IMAP servers tend to be better implemented.

The **poplib** module provides two classes:

```python
class poplib.POP3(host[, port[, timeout]])
```

This class implements the actual POP3 protocol. The connection is created when the instance is initialized. If `port` is omitted, the standard POP3 port (110) is used. The optional `timeout` parameter specifies a timeout in seconds for the connection attempt (if not specified, the global default timeout setting will be used).

*Changed in version 2.6: timeout was added.*

```python
class poplib.POP3_SSL(host[, port[, keyfile[, certfile]]])
```

This is a subclass of **POP3** that connects to the server over an SSL encrypted socket. If `port` is not specified, 995, the standard
POP3-over-SSL port is used. *keyfile* and *certfile* are also optional - they can contain a PEM formatted private key and certificate chain file for the SSL connection.

*New in version 2.4.*

One exception is defined as an attribute of the *poplib* module:

```python
exception poplib.error_proto
```

Exception raised on any errors from this module (errors from *socket* module are not caught). The reason for the exception is passed to the constructor as a string.

**See also:**

**Module imaplib**

The standard Python IMAP module.

**Frequently Asked Questions About Fetchmail**

The FAQ for the *fetchmail* POP/IMAP client collects information on POP3 server variations and RFC noncompliance that may be useful if you need to write an application based on the POP protocol.
All POP3 commands are represented by methods of the same name, in lower-case; most return the response text sent by the server.

An **POP3** instance has the following methods:

**POP3**. `set_debuglevel(level)`
Set the instance’s debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

**POP3**. `getwelcome()`
Returns the greeting string sent by the POP3 server.

**POP3**. `user(username)`
Send user command, response should indicate that a password is required.

**POP3**. `pass_(password)`
Send password, response includes message count and mailbox size. Note: the mailbox on the server is locked until `quit()` is called.

**POP3**. `apop(user, secret)`
Use the more secure APOP authentication to log into the POP3 server.

**POP3**. `rpop(user)`
Use RPOP authentication (similar to UNIX r-commands) to log
into POP3 server.

**POP3.** *stat()*
Get mailbox status. The result is a tuple of 2 integers: \((\text{message count}, \text{mailbox size})\).

**POP3.** *list([which])*
Request message list, result is in the form \((\text{response}, [\text{'mesg_num octets'}, ...], \text{octets})\). If *which* is set, it is the message to list.

**POP3.** *retr*(which)
Retrieve whole message number *which*, and set its seen flag. Result is in form \((\text{response}, [\text{'line'}, ...], \text{octets})\).

**POP3.** *dele*(which)
Flag message number *which* for deletion. On most servers deletions are not actually performed until QUIT (the major exception is Eudora QPOP, which deliberately violates the RFCs by doing pending deletes on any disconnect).

**POP3.** *rset()*
Remove any deletion marks for the mailbox.

**POP3.** *noop()*
Do nothing. Might be used as a keep-alive.

**POP3.** *quit()*
Signoff: commit changes, unlock mailbox, drop connection.

**POP3.** *top*(which, howmuch)*
Retrieves the message header plus *howmuch* lines of the message after the header of message number *which*. Result is in form \((\text{response}, [\text{'line'}, ...], \text{octets})\).

The POP3 TOP command this method uses, unlike the RETR
command, doesn’t set the message’s seen flag; unfortunately, TOP is poorly specified in the RFCs and is frequently broken in off-brand servers. Test this method by hand against the POP3 servers you will use before trusting it.

**POP3.uid1([which])**

Return message digest (unique id) list. If `which` is specified, result contains the unique id for that message in the form `response mesgnum uid`, otherwise result is list `(response, ['mesgnum uid', ...], octets)`.

Instances of **POP3_SSL** have no additional methods. The interface of this subclass is identical to its parent.
20.9.2. POP3 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```python
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print j
```

At the end of the module, there is a test section that contains a more extensive example of usage.
This module defines three classes, IMAP4, IMAP4_SSL and IMAP4_stream, which encapsulate a connection to an IMAP4 server and implement a large subset of the IMAP4rev1 client protocol as defined in RFC 2060. It is backward compatible with IMAP4 (RFC 1730) servers, but note that the STATUS command is not supported in IMAP4.

Three classes are provided by the imaplib module, IMAP4 is the base class:

```python
class imaplib.IMAP4([host[, port]])
```

This class implements the actual IMAP4 protocol. The connection is created and protocol version (IMAP4 or IMAP4rev1) is determined when the instance is initialized. If host is not specified, '' (the local host) is used. If port is omitted, the standard IMAP4 port (143) is used.

Three exceptions are defined as attributes of the IMAP4 class:

```python
exception IMAP4.error
```

Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

```python
exception IMAP4.abort
```

IMAP4 server errors cause this exception to be raised. This is a sub-class of IMAP4.error. Note that closing the instance and instantiating a new one will usually allow recovery from this
exception.

**exception IMAP4.readonly**

This exception is raised when a writable mailbox has its status changed by the server. This is a sub-class of IMAP4.error. Some other client now has write permission, and the mailbox will need to be re-opened to re-obtain write permission.

There’s also a subclass for secure connections:

```python
class imaplib.IMAP4_SSL([host[, port[, keyfile[, certfile]]]])
```

This is a subclass derived from IMAP4 that connects over an SSL encrypted socket (to use this class you need a socket module that was compiled with SSL support). If `host` is not specified, `'` (the local host) is used. If `port` is omitted, the standard IMAP4-over-SSL port (993) is used. `keyfile` and `certfile` are also optional - they can contain a PEM formatted private key and certificate chain file for the SSL connection.

The second subclass allows for connections created by a child process:

```python
class imaplib.IMAP4_stream(command)
```

This is a subclass derived from IMAP4 that connects to the stdin/stdout file descriptors created by passing `command` to `os.popen2()`.

*New in version 2.3.*

The following utility functions are defined:

```python
imaplib/Internaldate2tuple(datestr)
```

Parse an IMAP4 INTERNALDATE string and return corresponding local time. The return value is a `time.struct_time` instance or
None if the string has wrong format.

imaplib. **Int2AP**(num)

Converts an integer into a string representation using characters from the set [A .. P].

imaplib. **ParseFlags**(flagstr)

Converts an IMAP4 **FLAGS** response to a tuple of individual flags.

imaplib. **Time2Internaldate**(date_time)

Convert `date_time` to an IMAP4 **INTERNALDATE** representation. The return value is a string in the form: "DD-Mmm-YYYY HH:MM:SS +HHMM" (including double-quotes). The `date_time` argument can be a number (int or float) representing seconds since epoch (as returned by `time.time()`), a 9-tuple representing local time (as returned by `time.localtime()`), or a double-quoted string. In the last case, it is assumed to already be in the correct format.

Note that IMAP4 message numbers change as the mailbox changes; in particular, after an **EXPUNGE** command performs deletions the remaining messages are renumbered. So it is highly advisable to use UIDs instead, with the UID command.

At the end of the module, there is a test section that contains a more extensive example of usage.

**See also:** Documents describing the protocol, and sources and binaries for servers implementing it, can all be found at the University of Washington’s **IMAP Information Center** (https://www.washington.edu/imap/).
20.10.1. IMAP4 Objects

All IMAP4rev1 commands are represented by methods of the same name, either upper-case or lower-case.

All arguments to commands are converted to strings, except for AUTHENTICATE, and the last argument to APPEND which is passed as an IMAP4 literal. If necessary (the string contains IMAP4 protocol-sensitive characters and isn’t enclosed with either parentheses or double quotes) each string is quoted. However, the password argument to the LOGIN command is always quoted. If you want to avoid having an argument string quoted (eg: the flags argument to STORE) then enclose the string in parentheses (eg: r'("Deleted")').

Each command returns a tuple: (type, [data, ...]) where type is usually 'OK' or 'NO', and data is either the text from the command response, or mandated results from the command. Each data is either a string, or a tuple. If a tuple, then the first part is the header of the response, and the second part contains the data (ie: ‘literal’ value).

The message_set options to commands below is a string specifying one or more messages to be acted upon. It may be a simple message number ('1'), a range of message numbers ('2:4'), or a group of non-contiguous ranges separated by commas ('1:3,6:9'). A range can contain an asterisk to indicate an infinite upper bound ('3:*').

An IMAP4 instance has the following methods:

IMAP4. append(mailbox, flags, date_time, message)
    Append message to named mailbox.
**IMAP4.authenticate**(*mechanism, authobject*)

Authenticate command — requires response processing.

*mechanism* specifies which authentication mechanism is to be used - it should appear in the instance variable `capabilities` in the form `AUTH=mechanism`.

*authobject* must be a callable object:

```python
data = authobject(response)
```

It will be called to process server continuation responses. It should return `data` that will be encoded and sent to server. It should return `None` if the client abort response `*` should be sent instead.

**IMAP4.check()**

Checkpoint mailbox on server.

**IMAP4.close()**

Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before `LOGOUT`.

**IMAP4.copy**(*message_set, new_mailbox*)

Copy `message_set` messages onto end of `new_mailbox`.

**IMAP4.create**(*mailbox*)

Create new mailbox named `mailbox`.

**IMAP4.delete**(*mailbox*)

Delete old mailbox named `mailbox`.

**IMAP4.deleteacl**(*mailbox, who*)

Delete the ACLs (remove any rights) set for who on mailbox.
New in version 2.4.

**IMAP4. expunge()**
Permanently remove deleted items from selected mailbox. Generates an **EXPUNGE** response for each deleted message. Returned data contains a list of **EXPUNGE** message numbers in order received.

**IMAP4. fetch(message_set, message_parts)**
Fetch (parts of) messages. *message_parts* should be a string of message part names enclosed within parentheses, eg: "(UID BODY[TEXT])". Returned data are tuples of message part envelope and data.

**IMAP4. getacl(mailbox)**
Get the **ACLs** for *mailbox*. The method is non-standard, but is supported by the **cyrus** server.

**IMAP4. getannotation(mailbox, entry, attribute)**
Retrieve the specified **ANNOTATIONs** for *mailbox*. The method is non-standard, but is supported by the **cyrus** server.

New in version 2.5.

**IMAP4. getquota(root)**
Get the **quota** root's resource usage and limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

New in version 2.3.

**IMAP4. getquotaroot(mailbox)**
Get the list of **quota roots** for the named *mailbox*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

New in version 2.3.
**IMAP4. list([directory[, pattern]])**

List mailbox names in `directory` matching `pattern`. `directory` defaults to the top-level mail folder, and `pattern` defaults to match anything. Returned data contains a list of `LIST` responses.

**IMAP4. login(user, password)**

Identify the client using a plaintext password. The `password` will be quoted.

**IMAP4. login_cram_md5(user, password)**

Force use of CRAM-MD5 authentication when identifying the client to protect the password. Will only work if the server `CAPABILITY` response includes the phrase `AUTH=CRAM-MD5`.

*New in version 2.3.*

**IMAP4. logout()**

Shutdown connection to server. Returns server `BYE` response.

**IMAP4. lsub([directory[, pattern]])**

List subscribed mailbox names in `directory` matching `pattern`. `directory` defaults to the top level directory and `pattern` defaults to match any mailbox. Returned data are tuples of message part envelope and data.

**IMAP4. myrights(mailbox)**

Show my ACLs for a mailbox (i.e. the rights that I have on mailbox).

*New in version 2.4.*

**IMAP4. namespace()**

Returns IMAP namespaces as defined in RFC2342.

*New in version 2.3.*
**IMAP4.** `noop()`  
Send **NOOP** to server.

**IMAP4.** `open(host, port)`  
Opens socket to *port* at *host*. This method is implicitly called by the `IMAP4` constructor. The connection objects established by this method will be used in the `IMAP4.read()`, `IMAP4.readline()`, `IMAP4.send()`, and `IMAP4.shutdown()` methods. You may override this method.

**IMAP4.** `partial(message_num, message_part, start, length)`  
Fetch truncated part of a message. Returned data is a tuple of message part envelope and data.

**IMAP4.** `proxyauth(user)`  
Assume authentication as *user*. Allows an authorised administrator to proxy into any user's mailbox.

*New in version 2.3.*

**IMAP4.** `read(size)`  
Reads *size* bytes from the remote server. You may override this method.

**IMAP4.** `readline()`  
Reads one line from the remote server. You may override this method.

**IMAP4.** `recent()`  
Prompt server for an update. Returned data is *None* if no new messages, else value of **RECENT** response.

**IMAP4.** `rename(oldmailbox, newmailbox)`  
Rename mailbox named *oldmailbox* to *newmailbox*. 
**IMAP4. response(code)**

Return data for response code if received, or `None`. Returns the given code, instead of the usual type.

**IMAP4. search(charset, criterion[, ...])**

Search mailbox for matching messages. `charset` may be `None`, in which case no `CHARSET` will be specified in the request to the server. The IMAP protocol requires that at least one criterion be specified; an exception will be raised when the server returns an error.

Example:

```python
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

**IMAP4. select([mailbox[, readonly]])**

Select a mailbox. Returned data is the count of messages in `mailbox` (EXISTS response). The default `mailbox` is `"INBOX"`. If the `readonly` flag is set, modifications to the mailbox are not allowed.

**IMAP4. send(data)**

Sends `data` to the remote server. You may override this method.

**IMAP4. setacl(mailbox, who, what)**

Set an ACL for `mailbox`. The method is non-standard, but is supported by the `cyrus` server.

**IMAP4. setannotation(mailbox, entry, attribute[, ...])**

Set ANNOTATIONS for `mailbox`. The method is non-standard, but is supported by the `cyrus` server.
New in version 2.5.

**IMAP4. setquota(root, limits)**

Set the quota root's resource limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

New in version 2.3.

**IMAP4. shutdown()**

Close connection established in open. This method is implicitly called by **IMAP4.logout()**. You may override this method.

**IMAP4. socket()**

Returns socket instance used to connect to server.

**IMAP4. sort(sort_criteria, charset, search_criterion[, ...])**

The sort command is a variant of search with sorting semantics for the results. Returned data contains a space separated list of matching message numbers.

Sort has two arguments before the search_criterion argument(s); a parenthesized list of sort_criteria, and the searching charset. Note that unlike search, the searching charset argument is mandatory. There is also a uid sort command which corresponds to sort the way that uid search corresponds to search. The sort command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the numbers of matching messages.

This is an IMAP4rev1 extension command.

**IMAP4. status(mailbox, names)**

Request named status conditions for mailbox.
**IMAP4.store**(*message_set, command, flag_list*)

Alters flag dispositions for messages in mailbox. *command* is specified by section 6.4.6 of **RFC 2060** as being one of “FLAGS”, “+FLAGS”, or “-FLAGS”, optionally with a suffix of ”.SILENT”.

For example, to set the delete flag on all messages:

```python
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    M.store(num, '+FLAGS', '\Deleted')
M.expunge()
```

**IMAP4.subscribe**(*mailbox*)

Subscribe to new mailbox.

**IMAP4.thread**(*threading_algorithm, charset, search_criterion[, ...]*)

The **thread** command is a variant of **search** with threading semantics for the results. Returned data contains a space separated list of thread members.

Thread members consist of zero or more messages numbers, delimited by spaces, indicating successive parent and child.

Thread has two arguments before the *search_criterion* argument(s); a **threading_algorithm**, and the searching **charset**. Note that unlike **search**, the searching **charset** argument is mandatory. There is also a **uid thread** command which corresponds to **thread** the way that **uid search** corresponds to **search**. The **thread** command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the matching messages threaded according to the specified threading algorithm.

This is an **IMAP4rev1** extension command.
New in version 2.4.

**IMAP4. uid**(command, arg[, ...])

Execute command args with messages identified by UID, rather than message number. Returns response appropriate to command. At least one argument must be supplied; if none are provided, the server will return an error and an exception will be raised.

**IMAP4. unsubscribe**(mailbox)

Unsubscribe from old mailbox.

**IMAP4. xatom**(name[, arg[, ...]])

Allow simple extension commands notified by server in `CAPABILITY` response.

Instances of **IMAP4_SSL** have just one additional method:

**IMAP4_SSL. ssl()**

Returns SSLObj instance used for the secure connection with the server.

The following attributes are defined on instances of **IMAP4**:

**IMAP4. PROTOCOL_VERSION**

The most recent supported protocol in the `CAPABILITY` response from the server.

**IMAP4. debug**

Integer value to control debugging output. The initialize value is taken from the module variable `Debug`. Values greater than three trace each command.
20.10.2. IMAP4 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```python
import getpass, imaplib

M = imaplib.IMAP4()
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search('None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print 'Message %s
%s
' % (num, data[0][1])
M.close()
M.logout()
```
This module defines the class `NNTP` which implements the client side of the NNTP protocol. It can be used to implement a news reader or poster, or automated news processors. For more information on NNTP (Network News Transfer Protocol), see Internet RFC 977.

Here are two small examples of how it can be used. To list some statistics about a newsgroup and print the subjects of the last 10 articles:

```python
>>> s = NNTP('news.gmane.org')
>>> resp, count, first, last, name = s.group('gmane.comp.python.committers')
>>> print 'Group', name, 'has', count, 'articles, range', first + '-' + last
>>> resp, subs = s.xhdr('subject', first + '-' + last)
>>> for id, sub in subs[-10:]:
...     print id, sub
... 1062 Re: Mercurial Status?
1063 Re: [python-committers] (Windows) buildbots on 3.x
1064 Re: Mercurial Status?
1065 Re: Mercurial Status?
1066 Python 2.6.6 status
1067 Commit Privileges for Ask Solem
1068 Re: Commit Privileges for Ask Solem
1069 Re: Commit Privileges for Ask Solem
1070 Re: Commit Privileges for Ask Solem
1071 2.6.6 rc 2
>>> s.quit()
'205 Bye!'
```

To post an article from a file (this assumes that the article has valid headers, and that you have right to post on the particular
The module itself defines the following items:

```python
class nntplib.NNTP(host[, port [, user[, password [, readermode [, usenetrc]]]]])
```

Return a new instance of the `NNTP` class, representing a connection to the NNTP server running on host `host`, listening at port `port`. The default `port` is 119. If the optional `user` and `password` are provided, or if suitable credentials are present in `/.netrc` and the optional flag `usenetrc` is true (the default), the `AUTHINFO USER` and `AUTHINFO PASS` commands are used to identify and authenticate the user to the server. If the optional flag `readermode` is true, then a `mode reader` command is sent before authentication is performed. Reader mode is sometimes necessary if you are connecting to an NNTP server on the local machine and intend to call reader-specific commands, such as `group`. If you get unexpected `NNTPPermanentErrors`, you might need to set `readermode`. `readermode` defaults to `None`. `usenetrc` defaults to `True`.

*Changed in version 2.4:* `usenetrc` argument added.

```exception nntplib.NNTPError```

Derived from the standard exception `Exception`, this is the base class for all exceptions raised by the `nntplib` module.

```exception nntplib.NNTPReplyError```

Exception raised when an unexpected reply is received from the
server. For backwards compatibility, the exception `error_reply` is equivalent to this class.

**exception** `nntplib.NNTPTemporaryError`  
Exception raised when an error code in the range 400–499 is received. For backwards compatibility, the exception `error_temp` is equivalent to this class.

**exception** `nntplib.NNTPPermanentError`  
Exception raised when an error code in the range 500–599 is received. For backwards compatibility, the exception `error_perm` is equivalent to this class.

**exception** `nntplib.NNTPProtocolError`  
Exception raised when a reply is received from the server that does not begin with a digit in the range 1–5. For backwards compatibility, the exception `error_proto` is equivalent to this class.

**exception** `nntplib.NNTPDataError`  
Exception raised when there is some error in the response data. For backwards compatibility, the exception `error_data` is equivalent to this class.
20.11.1. NNTP Objects

NNTP instances have the following methods. The response that is returned as the first item in the return tuple of almost all methods is the server’s response: a string beginning with a three-digit code. If the server’s response indicates an error, the method raises one of the above exceptions.

NNTP.getwelcome()
Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

NNTP.set_debuglevel(level)
Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request or response. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the connection (including message text).

NNTP.newgroups(date, time[, file])
Send a NEWGROUPS command. The date argument should be a string of the form 'yymmd' indicating the date, and time should be a string of the form 'hhmmss' indicating the time. Return a pair (response, groups) where groups is a list of group names that are new since the given date and time. If the file parameter is supplied, then the output of the NEWGROUPS command is stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling write() on it to store the lines of the command
output. If file is supplied, then the returned list is an empty list.

NNTP. **newnews**(*group*, *date*, *time*, [file])
Send a **newnews** command. Here, *group* is a group name or '*', and *date* and *time* have the same meaning as for **newgroups**(). Return a pair (response, articles) where articles is a list of message ids. If the file parameter is supplied, then the output of the **newnews** command is stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling **write()** on it to store the lines of the command output. If file is supplied, then the returned list is an empty list.

NNTP. **list**([file])
Send a **list** command. Return a pair (response, list) where list is a list of tuples. Each tuple has the form (group, last, first, flag), where *group* is a group name, *last* and *first* are the last and first article numbers (as strings), and *flag* is 'y' if posting is allowed, 'n' if not, and 'm' if the newsgroup is moderated. (Note the ordering: last, first.) If the file parameter is supplied, then the output of the **list** command is stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling **write()** on it to store the lines of the command output. If file is supplied, then the returned list is an empty list.

NNTP. **descriptions**(*grouppattern*)
Send a **list** **newsgroups** command, where *grouppattern* is a wildmat string as specified in RFC2980 (it's essentially the same as DOS or UNIX shell wildcard strings). Return a pair (response, list), where list is a list of tuples containing (name, title).

*New in version 2.4.*
**NNTP. description**(group)

Get a description for a single group *group*. If more than one group matches (if ‘group’ is a real wildmat string), return the first match. If no group matches, return an empty string.

This elides the response code from the server. If the response code is needed, use descriptions().

*New in version 2.4.*

**NNTP. group**(name)

Send a GROUP command, where *name* is the group name. Return a tuple (response, count, first, last, name) where count is the (estimated) number of articles in the group, first is the first article number in the group, last is the last article number in the group, and name is the group name. The numbers are returned as strings.

**NNTP. help**([file])

Send a HELP command. Return a pair (response, list) where list is a list of help strings. If the file parameter is supplied, then the output of the HELP command is stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling write() on it to store the lines of the command output. If file is supplied, then the returned list is an empty list.

**NNTP. stat**(id)

Send a STAT command, where id is the message id (enclosed in '<' and '>') or an article number (as a string). Return a triple (response, number, id) where number is the article number (as a string) and id is the message id (enclosed in '<' and '>').
NNTP. **next()**
Send a NEXT command. Return as for **stat()**.

NNTP. **last()**
Send a LAST command. Return as for **stat()**.

NNTP. **head(id)**
Send a HEAD command, where *id* has the same meaning as for **stat()**. Return a tuple (**response**, **number**, **id**, **list**) where the first three are the same as for **stat()**, and **list** is a list of the article's headers (an uninterpreted list of lines, without trailing newlines).

NNTP. **body(id[, file])**
Send a BODY command, where *id* has the same meaning as for **stat()**. If the file parameter is supplied, then the body is stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling **write()** on it to store the lines of the body. Return as for head(). If file is supplied, then the returned **list** is an empty list.

NNTP. **article(id)**
Send an ARTICLE command, where *id* has the same meaning as for **stat()**. Return as for **head()**.

NNTP. **slave()**
Send a SLAVE command. Return the server’s response.

NNTP. **xhdr(header, string[, file])**
Send an XHDR command. This command is not defined in the RFC but is a common extension. The **header** argument is a header keyword, e.g. 'subject'. The **string** argument should have the form 'first-last' where first and last are the first
last article numbers to search. Return a pair \((\text{response}, \text{list})\), where \(\text{list}\) is a list of pairs \((\text{id}, \text{text})\), where \(\text{id}\) is an article number (as a string) and \(\text{text}\) is the text of the requested header for that article. If the \(\text{file}\) parameter is supplied, then the output of the \texttt{XHDR}\ command is stored in a file. If \(\text{file}\) is a string, then the method will open a file object with that name, write to it then close it. If \(\text{file}\) is a file object, then it will start calling \texttt{write()}\ on it to store the lines of the command output. If \(\text{file}\) is supplied, then the returned \(\text{list}\) is an empty list.

**NNTP.\ post(\text{file})**

Post an article using the \texttt{POST}\ command. The \(\text{file}\) argument is an open file object which is read until EOF using its \texttt{readline()}\ method. It should be a well-formed news article, including the required headers. The \texttt{post()}\ method automatically escapes lines beginning with \(\text{.}\).

**NNTP.\ ihave(\text{id}, \text{file})**

Send an \texttt{IHAVE}\ command. \(\text{id}\) is a message id (enclosed in \('<'\) and \('>')\). If the response is not an error, treat \(\text{file}\) exactly as for the \texttt{post()}\ method.

**NNTP.\ date()**

Return a triple \((\text{response}, \text{date}, \text{time})\), containing the current date and time in a form suitable for the \texttt{newnews()}\ and \texttt{newgroups()}\ methods. This is an optional NNTP extension, and may not be supported by all servers.

**NNTP.\ xgtitle(\text{name}[\text[, }\text{file}])**

Process an \texttt{XGTITLE}\ command, returning a pair \((\text{response}, \text{list})\), where \(\text{list}\) is a list of tuples containing \((\text{name}, \text{title})\). If the \(\text{file}\) parameter is supplied, then the output of the \texttt{XGTITLE}\ command is stored in a file. If \(\text{file}\) is a string, then the method will
open a file object with that name, write to it then close it. If file is a file object, then it will start calling write() on it to store the lines of the command output. If file is supplied, then the returned list is an empty list. This is an optional NNTP extension, and may not be supported by all servers.

RFC2980 says “It is suggested that this extension be deprecated”. Use descriptions() or description() instead.

NNTP. xover(start, end[, file])

Return a pair (resp, list). list is a list of tuples, one for each article in the range delimited by the start and end article numbers. Each tuple is of the form (article number, subject, poster, date, id, references, size, lines). If the file parameter is supplied, then the output of the XOVER command is stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling write() on it to store the lines of the command output. If file is supplied, then the returned list is an empty list. This is an optional NNTP extension, and may not be supported by all servers.

NNTP. xpath(id)

Return a pair (resp, path), where path is the directory path to the article with message ID id. This is an optional NNTP extension, and may not be supported by all servers.

NNTP. quit()

Send a QUIT command and close the connection. Once this method has been called, no other methods of the NNTP object should be called.
The `smtplib` module defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon. For details of SMTP and ESMTP operation, consult RFC 821 (Simple Mail Transfer Protocol) and RFC 1869 (SMTP Service Extensions).

```python
class smtplib.SMTP([host[, port[, local_hostname[, timeout]]]])

An SMTP instance encapsulates an SMTP connection. It has methods that support a full repertoire of SMTP and ESMTP operations. If the optional host and port parameters are given, the SMTP `connect()` method is called with those parameters during initialization. If specified, `local_hostname` is used as the FQDN of the local host in the HELO/EHLO command. Otherwise, the local hostname is found using `socket.getfqdn()`. If the `connect()` call returns anything other than a success code, an `SMTPConnectError` is raised. The optional `timeout` parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). If the timeout expires, `socket.timeout` is raised.

For normal use, you should only require the initialization/connect, `sendmail()`, and `quit()` methods. An example is included below.

Changed in version 2.6: timeout was added.

```
An `SMTP_SSL` instance behaves exactly the same as instances of `SMTP`. `SMTP_SSL` should be used for situations where SSL is required from the beginning of the connection and using `starttls()` is not appropriate. If `host` is not specified, the local host is used. If `port` is omitted, the standard SMTP-over-SSL port (465) is used. `local_hostname` has the same meaning as it does for the `SMTP` class. `keyfile` and `certfile` are also optional, and can contain a PEM formatted private key and certificate chain file for the SSL connection. The optional `timeout` parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). If the timeout expires, `socket.timeout` is raised.

New in version 2.6.

```
class smtplib.LMTP([host[, port[, local_hostname]]])
```

The LMTP protocol, which is very similar to ESMTP, is heavily based on the standard SMTP client. It's common to use Unix sockets for LMTP, so our `connect()` method must support that as well as a regular host:port server. `local_hostname` has the same meaning as it does for the `SMTP` class. To specify a Unix socket, you must use an absolute path for `host`, starting with a ‘/’.

Authentication is supported, using the regular SMTP mechanism. When using a Unix socket, LMTP generally don’t support or require any authentication, but your mileage might vary.

New in version 2.6.

A nice selection of exceptions is defined as well:

```
exception smtplib.SMTPException
```

The base exception class for all the other exceptions provided by
this module.

```

exception smtplib.SMTPServerDisconnected

This exception is raised when the server unexpectedly disconnects, or when an attempt is made to use the SMTP instance before connecting it to a server.

exception smtplib.SMTPResponseException

Base class for all exceptions that include an SMTP error code. These exceptions are generated in some instances when the SMTP server returns an error code. The error code is stored in the smtp_code attribute of the error, and the smtp_error attribute is set to the error message.

exception smtplib.SMTPSenderRefused

Sender address refused. In addition to the attributes set by on all SMTPResponseException exceptions, this sets ‘sender’ to the string that the SMTP server refused.

exception smtplib.SMTPRecipientsRefused

All recipient addresses refused. The errors for each recipient are accessible through the attribute recipients, which is a dictionary of exactly the same sort as SMTP.sendmail() returns.

exception smtplib.SMTPDataError

The SMTP server refused to accept the message data.

exception smtplib.SMTPConnectError

Error occurred during establishment of a connection with the server.

exception smtplib.SMTPHeloError

The server refused our HELO message.

exception smtplib.SMTPAuthenticationError
```
SMTP authentication went wrong. Most probably the server didn’t accept the username/password combination provided.

See also:

RFC 821 - Simple Mail Transfer Protocol
Protocol definition for SMTP. This document covers the model, operating procedure, and protocol details for SMTP.

RFC 1869 - SMTP Service Extensions
Definition of the ESMTP extensions for SMTP. This describes a framework for extending SMTP with new commands, supporting dynamic discovery of the commands provided by the server, and defines a few additional commands.
20.12.1. SMTP Objects

An SMTP instance has the following methods:

SMTP.set_debuglevel(level)
Set the debug output level. A true value for level results in debug messages for connection and for all messages sent to and received from the server.

SMTP.docmd(cmd[, argstring])
Send a command cmd to the server. The optional argument argstring is simply concatenated to the command, separated by a space.

This returns a 2-tuple composed of a numeric response code and the actual response line (multiline responses are joined into one long line.)

In normal operation it should not be necessary to call this method explicitly. It is used to implement other methods and may be useful for testing private extensions.

If the connection to the server is lost while waiting for the reply, SMTPServerDisconnected will be raised.

SMTP.connect([host[, port]])
Connect to a host on a given port. The defaults are to connect to the local host at the standard SMTP port (25). If the hostname ends with a colon (':') followed by a number, that suffix will be stripped off and the number interpreted as the port number to use. This method is automatically invoked by the constructor if a host is specified during instantiation. Returns a 2-tuple of the response code and message sent by the server in its connection
response.

**SMTP. `helo([hostname])`**

Identify yourself to the SMTP server using HELO. The hostname argument defaults to the fully qualified domain name of the local host. The message returned by the server is stored as the `helo_resp` attribute of the object.

In normal operation it should not be necessary to call this method explicitly. It will be implicitly called by the `sendmail()` when necessary.

**SMTP. `ehlo([hostname])`**

Identify yourself to an ESMTP server using EHLO. The hostname argument defaults to the fully qualified domain name of the local host. Examine the response for ESMTP option and store them for use by `has_extn()`. Also sets several informational attributes: the message returned by the server is stored as the `ehlo_resp` attribute, `does_esmtp` is set to true or false depending on whether the server supports ESMTP, and `esmtp_features` will be a dictionary containing the names of the SMTP service extensions this server supports, and their parameters (if any).

Unless you wish to use `has_extn()` before sending mail, it should not be necessary to call this method explicitly. It will be implicitly called by `sendmail()` when necessary.

**SMTP. `ehlo_or_helo_if_needed()`**

This method call `ehlo()` and or `helo()` if there has been no previous EHLO or HELO command this session. It tries ESMTP EHLO first.

`SMTPHeloError`
The server didn’t reply properly to the **HELO** greeting.

*New in version 2.6.*

**SMTP.** **has_extn**(*name*)

Return **True** if *name* is in the set of SMTP service extensions returned by the server, **False** otherwise. Case is ignored.

**SMTP.** **verify**(*address*)

Check the validity of an address on this server using SMTP **VRFY**. Returns a tuple consisting of code 250 and a full **RFC 822** address (including human name) if the user address is valid. Otherwise returns an SMTP error code of 400 or greater and an error string.

**Note:** Many sites disable SMTP **VRFY** in order to foil spammers.

**SMTP.** **login**(*user, password*)

Log in on an SMTP server that requires authentication. The arguments are the username and the password to authenticate with. If there has been no previous **EHLO** or **HELO** command this session, this method tries ESMTP **EHLO** first. This method will return normally if the authentication was successful, or may raise the following exceptions:

**SMTPHeloError**

The server didn’t reply properly to the **HELO** greeting.

**SMTPAuthenticationError**

The server didn’t accept the username/password combination.

**SMTPException**

No suitable authentication method was found.
SMTP\cdot\texttt{starttls}([\textit{keyfile}[[, \textit{certfile}]]])

Put the SMTP connection in TLS (Transport Layer Security) mode. All SMTP commands that follow will be encrypted. You should then call \texttt{ehlo()} again.

If \textit{keyfile} and \textit{certfile} are provided, these are passed to the \texttt{socket} module's \texttt{ssl()} function.

If there has been no previous \texttt{EHLO} or \texttt{HELO} command this session, this method tries ESMTP \texttt{EHLO} first.

\textit{Changed in version 2.6.}

\texttt{SMTPHeloError}

The server didn't reply properly to the \texttt{HELO} greeting.

\texttt{SMTPException}

The server does not support the STARTTLS extension.

\textit{Changed in version 2.6.}

\texttt{RuntimeError}

SSL/TLS support is not available to your Python interpreter.

\texttt{SMTP\cdot\texttt{sendmail}(\textit{from\_addr}, \textit{to\_addrs}, \textit{msg}[[, \textit{mail\_options}, \textit{rcpt\_options}])}

Send mail. The required arguments are an RFC 822 from-address string, a list of RFC 822 to-address strings (a bare string will be treated as a list with 1 address), and a message string. The caller may pass a list of ESMTP options (such as \texttt{8bitmime}) to be used in \texttt{MAIL FROM} commands as \textit{mail\_options}. ESMTP options (such as \texttt{DSN} commands) that should be used with all \texttt{RCPT} commands can be passed as \textit{rcpt\_options}. (If you need to use different ESMTP options to different recipients you have to
use the low-level methods such as `mail()`, `rcpt()` and `data()` to send the message.)

**Note:** The `from_addr` and `to_addrs` parameters are used to construct the message envelope used by the transport agents. The SMTP does not modify the message headers in any way.

If there has been no previous `EHLO` or `HELO` command this session, this method tries ESMTP `EHLO` first. If the server does ESMTP, message size and each of the specified options will be passed to it (if the option is in the feature set the server advertises). If `EHLO` fails, `HELO` will be tried and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one recipient. Otherwise it will raise an exception. That is, if this method does not raise an exception, then someone should get your mail. If this method does not raise an exception, it returns a dictionary, with one entry for each recipient that was refused. Each entry contains a tuple of the SMTP error code and the accompanying error message sent by the server.

This method may raise the following exceptions:

**SMTPRecipientsRefused**

All recipients were refused. Nobody got the mail. The `recipients` attribute of the exception object is a dictionary with information about the refused recipients (like the one returned when at least one recipient was accepted).

**SMTPHeLoError**

The server didn’t reply properly to the `HELO` greeting.

**SMTPSenderRefused**

The server didn’t accept the `from_addr`.
SMTPDataError

The server replied with an unexpected error code (other than a refusal of a recipient).

Unless otherwise noted, the connection will be open even after an exception is raised.

SMTP.quit()

Terminate the SMTP session and close the connection. Return the result of the SMTP quit command.

Changed in version 2.6: Return a value.

Low-level methods corresponding to the standard SMTP/ESMTP commands HELP, RSET, NOOP, MAIL, RCPT, and DATA are also supported. Normally these do not need to be called directly, so they are not documented here. For details, consult the module code.
20.12.2. SMTP Example

This example prompts the user for addresses needed in the message envelope ('To' and 'From' addresses), and the message to be delivered. Note that the headers to be included with the message must be included in the message as entered; this example doesn’t do any processing of the RFC 822 headers. In particular, the ‘To’ and ‘From’ addresses must be included in the message headers explicitly.

```python
import smtplib

def prompt(prompt):
    return raw_input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs = prompt("To: ").split()
print "Enter message, end with ^D (Unix) or ^Z (Windows):"

# Add the From: and To: headers at the start!
msg = ("From: %s\nTo: %s\n\n" % (fromaddr, "", "
  %s\n" % (fromaddr, ", ".join(toaddrs))

while 1:
    try:
        line = raw_input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print "Message length is " + repr(len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

**Note:** In general, you will want to use the `email` package's features to construct an email message, which you can then
convert to a string and send via `sendmail()`; see email: Examples.
This module offers several classes to implement SMTP servers. One is a generic do-nothing implementation, which can be overridden, while the other two offer specific mail-sending strategies.
20.13.1. SMTPServer Objects

class smtpd.SMTPServer(localaddr, remoteaddr)

Create a new SMTPServer object, which binds to local address localaddr. It will treat remoteaddr as an upstream SMTP relayer. Both localaddr and remoteaddr should be a (host, port) tuple. The object inherits from asyncore.dispatcher, and so will insert itself into asyncore's event loop on instantiation.

process_message(peer, mailfrom, rcpttos, data)

Raise NotImplementedError exception. Override this in subclasses to do something useful with this message. Whatever was passed in the constructor as remoteaddr will be available as the _remoteaddr attribute. peer is the remote host's address, mailfrom is the envelope originator, rcpttos are the envelope recipients and data is a string containing the contents of the e-mail (which should be in RFC 2822 format).
20.13.2. DebuggingServer Objects

```python
class smtpd. DebuggingServer(localaddr, remoteaddr)
    Create a new debugging server. Arguments are as per
    SMTPServer. Messages will be discarded, and printed on stdout.
```
20.13.3. PureProxy Objects

class smtpd.PureProxy(localaddr, remoteaddr)

Create a new pure proxy server. Arguments are as per smtpd.SMTPServer. Everything will be relayed to remoteaddr. Note that running this has a good chance to make you into an open relay, so please be careful.
20.13.4. MailmanProxy Objects

```python
class smtpd.MailmanProxy(localaddr, remoteaddr)

Create a new pure proxy server. Arguments are as per SMTPServer. Everything will be relayed to remoteaddr, unless local mailman configurations knows about an address, in which case it will be handled via mailman. Note that running this has a good chance to make you into an open relay, so please be careful.
```
The `telnetlib` module provides a `Telnet` class that implements the Telnet protocol. See RFC 854 for details about the protocol. In addition, it provides symbolic constants for the protocol characters (see below), and for the telnet options. The symbolic names of the telnet options follow the definitions in `arpa/telnet.h`, with the leading `TELOPT_` removed. For symbolic names of options which are traditionally not included in `arpa/telnet.h`, see the module source itself.

The symbolic constants for the telnet commands are: IAC, DONT, DO, WONT, WILL, SE (Subnegotiation End), NOP (No Operation), DM (Data Mark), BRK (Break), IP (Interrupt process), AO (Abort output), AYT (Are You There), EC (Erase Character), EL (Erase Line), GA (Go Ahead), SB (Subnegotiation Begin).

```python
class telnetlib.Telnet([host[, port[, timeout]]])
```

`Telnet` represents a connection to a Telnet server. The instance is initially not connected by default; the `open()` method must be used to establish a connection. Alternatively, the host name and optional port number can be passed to the constructor, to, in which case the connection to the server will be established before the constructor returns. The optional `timeout` parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).

Do not reopen an already connected instance.
This class has many `read_*( )` methods. Note that some of them raise `EOFError` when the end of the connection is read, because they can return an empty string for other reasons. See the individual descriptions below.

*Changed in version 2.6: timeout was added.*

---

See also:

**RFC 854 - Telnet Protocol Specification**

- Definition of the Telnet protocol.
20.14.1. Telnet Objects

Telnet instances have the following methods:

Telnet. `read_until(expected[, timeout])`
Read until a given string, `expected`, is encountered or until `timeout` seconds have passed.

When no match is found, return whatever is available instead, possibly the empty string. Raise `EOFError` if the connection is closed and no cooked data is available.

Telnet. `read_all()`
Read all data until EOF; block until connection closed.

Telnet. `read_some()`
Read at least one byte of cooked data unless EOF is hit. Return `''` if EOF is hit. Block if no data is immediately available.

Telnet. `read_very_eager()`
Read everything that can be without blocking in I/O (eager).

Raise `EOFError` if connection closed and no cooked data available. Return `''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

Telnet. `read_eager()`
Read readily available data.

Raise `EOFError` if connection closed and no cooked data available. Return `''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

Telnet. `read_lazy()`
Process and return data already in the queues (lazy).

Raise `EOFError` if connection closed and no data available. Return `' ' if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

Telnet. `readVeryLazy()`
Return any data available in the cooked queue (very lazy).

Raise `EOFError` if connection closed and no data available. Return `' ' if no cooked data available otherwise. This method never blocks.

Telnet. `readSbData()`
Return the data collected between a SB/SE pair (suboption begin/end). The callback should access these data when it was invoked with a SE command. This method never blocks.

*New in version 2.3.*

Telnet. `open(host[, port[, timeout]])`
Connect to a host. The optional second argument is the port number, which defaults to the standard Telnet port (23). The optional `timeout` parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).

Do not try to reopen an already connected instance.

*Changed in version 2.6: timeout was added.*

Telnet. `msg(msg[, *args])`
Print a debug message when the debug level is \( \geq 0 \). If extra arguments are present, they are substituted in the message using the standard string formatting operator.
Telnet. **set_debuglevel**(*debuglevel*)

Set the debug level. The higher the value of *debuglevel*, the more debug output you get (on `sys.stdout`).

Telnet. **close**()

Close the connection.

Telnet. **get_socket**()

Return the socket object used internally.

Telnet. **fileno**()

Return the file descriptor of the socket object used internally.

Telnet. **write**(*buffer*)

Write a string to the socket, doubling any IAC characters. This can block if the connection is blocked. May raise `socket.error` if the connection is closed.

Telnet. **interact**()

Interaction function, emulates a very dumb Telnet client.

Telnet. **mt_interact**()

Multithreaded version of `interact()`.

Telnet. **expect**(*list[, timeout]*)

Read until one from a list of a regular expressions matches.

The first argument is a list of regular expressions, either compiled (`regex objects`) or uncompiled (strings). The optional second argument is a timeout, in seconds; the default is to block indefinitely.

Return a tuple of three items: the index in the list of the first regular expression that matches; the match object returned; and the text read up till and including the match.
If end of file is found and no text was read, raise `EOFError`. Otherwise, when nothing matches, return `(-1, None, text)` where `text` is the text received so far (may be the empty string if a timeout happened).

If a regular expression ends with a greedy match (such as `.*`) or if more than one expression can match the same input, the results are non-deterministic, and may depend on the I/O timing.

**Telnet.** `set_option_negotiation_callback(callback)`

Each time a telnet option is read on the input flow, this `callback` (if set) is called with the following parameters: `callback(telnet socket, command (DO/DONT/WILL/WONT), option)`. No other action is done afterwards by telnetlib.
20.14.2. Telnet Example

A simple example illustrating typical use:

```python
import getpass
import sys
import telnetlib

HOST = "localhost"
user = raw_input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)
	n.read_until("login: ")
	n.write(user + "\n")

if password:
    tnm.read_until("Password: ")
    tnm.write(password + "\n")


tn.write("ls\n")
	n.write("exit\n")

print tnm.read_all()
```
20.15. uuid — UUID objects according to RFC 4122

New in version 2.5.

This module provides immutable UUID objects (the UUID class) and the functions uuid1(), uuid3(), uuid4(), uuid5() for generating version 1, 3, 4, and 5 UUIDs as specified in RFC 4122.

If all you want is a unique ID, you should probably call uuid1() or uuid4(). Note that uuid1() may compromise privacy since it creates a UUID containing the computer's network address. uuid4() creates a random UUID.

```
class uuid.UUID([hex[, bytes[, bytes_le[, fields[, int[, version]]]]]]])
```

Create a UUID from either a string of 32 hexadecimal digits, a string of 16 bytes as the bytes argument, a string of 16 bytes in little-endian order as the bytes_le argument, a tuple of six integers (32-bit time_low, 16-bit time_mid, 16-bit time_hi_version, 8-bit clock_seq_hi_variant, 8-bit clock_seq_low, 48-bit node) as the fields argument, or a single 128-bit integer as the int argument. When a string of hex digits is given, curly braces, hyphens, and a URN prefix are all optional. For example, these expressions all yield the same UUID:

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes='\x12\x34\x56\x78'*4)
UUID(bytes_le='\x78\x56\x34\x12\x34\x12\x78\x56' + '
\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x56781)
UUID(int=0x12345678123456781234567812345678)
Exactly one of hex, bytes, bytes_le, fields, or int must be given. The version argument is optional; if given, the resulting UUID will have its variant and version number set according to RFC 4122, overriding bits in the given hex, bytes, bytes_le, fields, or int.

UUID instances have these read-only attributes:

UUID.**bytes**
The UUID as a 16-byte string (containing the six integer fields in big-endian byte order).

UUID.**bytes_le**
The UUID as a 16-byte string (with time_low, time_mid, and time_hi_version in little-endian byte order).

UUID.**fields**
A tuple of the six integer fields of the UUID, which are also available as six individual attributes and two derived attributes:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>time_low</td>
<td>the first 32 bits of the UUID</td>
</tr>
<tr>
<td>time_mid</td>
<td>the next 16 bits of the UUID</td>
</tr>
<tr>
<td>time_hi_version</td>
<td>the next 16 bits of the UUID</td>
</tr>
<tr>
<td>clock_seq_hi_variant</td>
<td>the next 8 bits of the UUID</td>
</tr>
<tr>
<td>clock_seq_low</td>
<td>the next 8 bits of the UUID</td>
</tr>
<tr>
<td>node</td>
<td>the last 48 bits of the UUID</td>
</tr>
<tr>
<td>time</td>
<td>the 60-bit timestamp</td>
</tr>
<tr>
<td>clock_seq</td>
<td>the 14-bit sequence number</td>
</tr>
</tbody>
</table>

UUID.**hex**
The UUID as a 32-character hexadecimal string.

UUID.**int**
The UUID as a 128-bit integer.
**UUID. urn**
The UUID as a URN as specified in RFC 4122.

**UUID. variant**
The UUID variant, which determines the internal layout of the UUID. This will be one of the integer constants `RESERVED_NCS`, `RFC_4122`, `RESERVED_MICROSOFT`, or `RESERVED_FUTURE`.

**UUID. version**
The UUID version number (1 through 5, meaningful only when the variant is `RFC_4122`).

The `uuid` module defines the following functions:

`uuid.getnode()`
Get the hardware address as a 48-bit positive integer. The first time this runs, it may launch a separate program, which could be quite slow. If all attempts to obtain the hardware address fail, we choose a random 48-bit number with its eighth bit set to 1 as recommended in RFC 4122. “Hardware address” means the MAC address of a network interface, and on a machine with multiple network interfaces the MAC address of any one of them may be returned.

`uuid.uuid1([node[, clock_seq]])`
Generate a UUID from a host ID, sequence number, and the current time. If `node` is not given, `getnode()` is used to obtain the hardware address. If `clock_seq` is given, it is used as the sequence number; otherwise a random 14-bit sequence number is chosen.

`uuid.uuid3(namespace, name)`
Generate a UUID based on the MD5 hash of a namespace identifier (which is a UUID) and a name (which is a string).
uuid. **uuid4()**
Generate a random UUID.

uuid. **uuid5(namespace, name)**
Generate a UUID based on the SHA-1 hash of a namespace identifier (which is a UUID) and a name (which is a string).

The **uuid** module defines the following namespace identifiers for use with **uuid3()** or **uuid5()**.

uuid. **NAMESPACE_DNS**
When this namespace is specified, the *name* string is a fully-qualified domain name.

uuid. **NAMESPACE_URL**
When this namespace is specified, the *name* string is a URL.

uuid. **NAMESPACE_OID**
When this namespace is specified, the *name* string is an ISO OID.

uuid. **NAMESPACE_X500**
When this namespace is specified, the *name* string is an X.500 DN in DER or a text output format.

The **uuid** module defines the following constants for the possible values of the **variant** attribute:

uuid. **RESERVED_NCS**
Reserved for NCS compatibility.

uuid. **RFC_4122**
Specifies the UUID layout given in RFC 4122.

uuid. **RESERVED_MICROSOFT**
Reserved for Microsoft compatibility.
**uuid RESERVED_FUTURE**

Reserved for future definition.

See also:

**RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace**

This specification defines a Uniform Resource Name namespace for UUIDs, the internal format of UUIDs, and methods of generating UUIDs.
Here are some examples of typical usage of the `uuid` module:

```python
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')

>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```
20.16. urlparse — Parse URLs into components

**Note:** The `urlparse` module is renamed to `urllib.parse` in Python 3. The `2to3` tool will automatically adapt imports when converting your sources to Python 3.

**Source code:** Lib/urlparse.py

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a “relative URL” to an absolute URL given a “base URL.”

The module has been designed to match the Internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes: `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `mailto`, `mms`, `news`, `nntp`, `prospero`, `rsync`, `rtsp`, `rtspu`, `sftp`, `shttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`.

*New in version 2.5:* Support for the `sftp` and `sips` schemes.

The `urlparse` module defines the following functions:

```
urlparse(urlstring[, scheme[, allow_fragments]])
```

Parse a URL into six components, returning a 6-tuple. This corresponds to the general structure of a URL: `scheme://netloc/path;parameters?query#fragment`. Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (for example, the network location is a single
string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the path component, which is retained if present. For example:

```python
>>> from urlparse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html', params='', query='', fragment='')
>>> o.scheme
'http'
>>> o.port
80
>>> o.geturl()
'http://www.cwi.nl:80/%7Eguido/Python.html'
```

Following the syntax specifications in RFC 1808, urlparse recognizes a netloc only if it is properly introduced by ‘//’. Otherwise the input is presumed to be a relative URL and thus to start with a path component.

```python
>>> from urlparse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html', params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html', params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='', query='', fragment='')
```

If the scheme argument is specified, it gives the default addressing scheme, to be used only if the URL does not specify one. The default value for this argument is the empty string.

If the allow fragments argument is false, fragment identifiers are not recognized and parsed as part of the preceding component, even if the URL's addressing scheme normally does support
them. The default value for this argument is `True`.

The return value is actually an instance of a subclass of `tuple`. This class has the following additional read-only convenience attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Index</th>
<th>Value</th>
<th>Value if not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheme</td>
<td>0</td>
<td>URL scheme specifier</td>
<td><code>scheme</code> parameter</td>
</tr>
<tr>
<td>netloc</td>
<td>1</td>
<td>Network location part</td>
<td>empty string</td>
</tr>
<tr>
<td>path</td>
<td>2</td>
<td>Hierarchical path</td>
<td>empty string</td>
</tr>
<tr>
<td>params</td>
<td>3</td>
<td>Parameters for last path element</td>
<td>empty string</td>
</tr>
<tr>
<td>query</td>
<td>4</td>
<td>Query component</td>
<td>empty string</td>
</tr>
<tr>
<td>fragment</td>
<td>5</td>
<td>Fragment identifier</td>
<td>empty string</td>
</tr>
<tr>
<td>username</td>
<td></td>
<td>User name</td>
<td><code>None</code></td>
</tr>
<tr>
<td>password</td>
<td></td>
<td>Password</td>
<td><code>None</code></td>
</tr>
<tr>
<td>hostname</td>
<td></td>
<td>Host name (lower case)</td>
<td><code>None</code></td>
</tr>
<tr>
<td>port</td>
<td></td>
<td>Port number as integer, if present</td>
<td><code>None</code></td>
</tr>
</tbody>
</table>

See section Results of `urlparse()` and `urlsplit()` for more information on the result object.

*Changed in version 2.5:* Added attributes to return value.

*Changed in version 2.7:* Added IPv6 URL parsing capabilities.

`urlparse.parse_qs(qs[, keep_blank_values[, strict_parsing]])`

Parse a query string given as a string argument (data of type `application/x-www-form-urlencoded`). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.
The optional argument `keep_blank_values` is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument `strict_parsing` is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

Use the `urllib.urlencode()` function to convert such dictionaries into query strings.

*New in version 2.6:* Copied from the `cgi` module.

```python
urlparse.parse_qs(qs[, keep_blank_values[, strict_parsing]])
```
Parse a query string given as a string argument (data of type `application/x-www-form-urlencoded`). Data are returned as a list of name, value pairs.

The optional argument `keep_blank_values` is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument `strict_parsing` is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

Use the `urllib.urlencode()` function to convert such lists of pairs into query strings.
*New in version 2.6:* Copied from the *cgi* module.

`urlparse. urlunparse(parts)`

Construct a URL from a tuple as returned by `urlparse()`. The `parts` argument can be any six-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a `?` with an empty query; the RFC states that these are equivalent).

`urlparse. urlsplit(urlstring[, scheme[, allow_fragments]])`

This is similar to `urlparse()`, but does not split the params from the URL. This should generally be used instead of `urlparse()` if the more recent URL syntax allowing parameters to be applied to each segment of the `path` portion of the URL (see RFC 2396) is wanted. A separate function is needed to separate the path segments and parameters. This function returns a 5-tuple: (addressing scheme, network location, path, query, fragment identifier).

The return value is actually an instance of a subclass of `tuple`. This class has the following additional read-only convenience attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Index</th>
<th>Value</th>
<th>Value if not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheme</td>
<td>0</td>
<td>URL scheme specifier</td>
<td><code>scheme</code> parameter</td>
</tr>
<tr>
<td>netloc</td>
<td>1</td>
<td>Network location part</td>
<td>empty string</td>
</tr>
<tr>
<td>path</td>
<td>2</td>
<td>Hierarchical path</td>
<td>empty string</td>
</tr>
<tr>
<td>query</td>
<td>3</td>
<td>Query component</td>
<td>empty string</td>
</tr>
<tr>
<td>fragment</td>
<td>4</td>
<td>Fragment identifier</td>
<td>empty string</td>
</tr>
<tr>
<td>username</td>
<td></td>
<td>User name</td>
<td><code>None</code></td>
</tr>
<tr>
<td>password</td>
<td></td>
<td>Password</td>
<td><code>None</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Host name (lower</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
<td>------------</td>
<td></td>
</tr>
<tr>
<td>hostname</td>
<td>case)</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>port</td>
<td>Port number as integer, if present</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

See section [Results of urlparse() and urlsplit()](#) for more information on the result object.

*New in version 2.2.*

*Changed in version 2.5:* Added attributes to return value.

**urlparse.**`urlunsplit(parts)`

Combine the elements of a tuple as returned by `urlsplit()` into a complete URL as a string. The *parts* argument can be any five-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).

*New in version 2.2.*

**urlparse.**`urljoin(base, url[, allow_fragments])`

Construct a full ("absolute") URL by combining a "base URL" (*base*) with another URL (*url*). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL. For example:

```python
>>> from urlparse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The *allow_fragments* argument has the same meaning and default as for `urlparse()`.
Note: If \textit{url} is an absolute URL (that is, starting with // or scheme://), the \textit{url}'s host name and/or scheme will be present in the result. For example:

\begin{verbatim}
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
\end{verbatim}

If you do not want that behavior, preprocess the \textit{url} with \texttt{urlsplit()} and \texttt{urlunsplit()}, removing possible \textit{scheme} and \textit{netloc} parts.

\texttt{urlparse.urldefrag(url)}

If \textit{url} contains a fragment identifier, returns a modified version of \textit{url} with no fragment identifier, and the fragment identifier as a separate string. If there is no fragment identifier in \textit{url}, returns \textit{url} unmodified and an empty string.

\textbf{See also:}

\textbf{RFC 3986 - Uniform Resource Identifiers}

This is the current standard (STD66). Any changes to \texttt{urlparse} module should conform to this. Certain deviations could be observed, which are mostly for backward compatibility purposes and for certain de-facto parsing requirements as commonly observed in major browsers.

\textbf{RFC 2732 - Format for Literal IPv6 Addresses in URL's.}

This specifies the parsing requirements of IPv6 URLs.

\textbf{RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax}

Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

\textbf{RFC 2368 - The mailto URL scheme.}
Parsing requirements for mailto URL schemes.

**RFC 1808 - Relative Uniform Resource Locators**

This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair number of “Abnormal Examples” which govern the treatment of border cases.

**RFC 1738 - Uniform Resource Locators (URL)**

This specifies the formal syntax and semantics of absolute URLs.
The result objects from the `urlparse()` and `urlsplit()` functions are subclasses of the `tuple` type. These subclasses add the attributes described in those functions, as well as provide an additional method:

```
ParseResult.geturl()
```

Return the re-combined version of the original URL as a string. This may differ from the original URL in that the scheme will always be normalized to lower case and empty components may be dropped. Specifically, empty parameters, queries, and fragment identifiers will be removed.

The result of this method is a fixpoint if passed back through the original parsing function:

```python
>>> import urlparse
>>> url = 'HTTP://www.Python.org/doc/#'

>>> r1 = urlparse.urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'

>>> r2 = urlparse.urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

*New in version 2.5.*

The following classes provide the implementations of the parse results:

```python
class urlparse.ParseResult(scheme, netloc, path, params, query, fragment)
```

Concrete class for `urlparse()` results.
class urlparse.SplitResult(scheme, netloc, path, query, fragment)

Concrete class for urllsplit() results.
20.17. **SocketServer — A framework for network servers**

**Note:** The `SocketServer` module has been renamed to `socketserver` in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

**Source code:** Lib/SocketServer.py

The `SocketServer` module simplifies the task of writing network servers.

There are four basic concrete server classes:

```python
class SocketServer.TCPServer(server_address, RequestHandlerClass, bind_and_activate=True):
    This uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. If `bind_and_activate` is true, the constructor automatically attempts to invoke `server_bind()` and `server_activate()`. The other parameters are passed to the `BaseServer` base class.

class SocketServer.UDPServer(server_address, RequestHandlerClass, bind_and_activate=True):
    This uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The parameters are the same as for `TCPServer`.

class SocketServer.UnixStreamServer(server_address, RequestHandlerClass, bind_and_activate=True)
class SocketServer.UnixDatagramServer(server_address, RequestHandlerClass, bind_and_activate=True)
```
These more infrequently used classes are similar to the TCP and UDP classes, but use Unix domain sockets; they’re not available on non-Unix platforms. The parameters are the same as for TCPServer.

These four classes process requests *synchronously*; each request must be completed before the next request can be started. This isn’t suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request; the ForkingMixIn and ThreadingMixIn mix-in classes can be used to support asynchronous behaviour.

Creating a server requires several steps. First, you must create a request handler class by subclassing the BaseRequestHandler class and overriding its handle() method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server’s address and the request handler class. Then call the handle_request() or serve_forever() method of the server object to process one or many requests. Finally, call server_close() to close the socket.

When inheriting from ThreadingMixIn for threaded connection behavior, you should explicitly declare how you want your threads to behave on an abrupt shutdown. The ThreadingMixIn class defines an attribute daemon_threads, which indicates whether or not the server should wait for thread termination. You should set the flag explicitly if you would like threads to behave autonomously; the default is False, meaning that Python will not exit until all threads created by ThreadingMixIn have exited.

Server classes have the same external methods and attributes, no matter what network protocol they use.
20.17.1. Server Creation Notes

There are five classes in an inheritance diagram, four of which represent synchronous servers of four types:

```
+------------------+
| TCPServer        |
+------------------+
<table>
<thead>
<tr>
<th>UnixStreamServer</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDPServer</td>
</tr>
</tbody>
</table>
+------------------+
```

Note that `UnixDatagramServer` derives from `UDPServer`, not from `UnixStreamServer` — the only difference between an IP and a Unix stream server is the address family, which is simply repeated in both Unix server classes.

```python
class SocketServer.ForkingMixIn
```

```python
class SocketServer.ThreadingMixIn
```

Forking and threading versions of each type of server can be created using these mix-in classes. For instance, `ThreadingUDPServer` is created as follows:

```python
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

The mix-in class comes first, since it overrides a method defined in `UDPServer`. Setting the various attributes also changes the behavior of the underlying server mechanism.
ForkingMixIn and the Forking classes mentioned below are only available on POSIX platforms that support fork().

class SocketServer.ForkingTCPServer
class SocketServer.ForkingUDPServer
class SocketServer.ThreadingTCPServer
class SocketServer.ThreadingUDPServer

These classes are pre-defined using the mix-in classes.

To implement a service, you must derive a class from BaseRequestHandler and redefine its handle() method. You can then run various versions of the service by combining one of the server classes with your request handler class. The request handler class must be different for datagram or stream services. This can be hidden by using the handler subclasses StreamRequestHandler or DatagramRequestHandler.

Of course, you still have to use your head! For instance, it makes no sense to use a forking server if the service contains state in memory that can be modified by different requests, since the modifications in the child process would never reach the initial state kept in the parent process and passed to each child. In this case, you can use a threading server, but you will probably have to use locks to protect the integrity of the shared data.

On the other hand, if you are building an HTTP server where all data is stored externally (for instance, in the file system), a synchronous class will essentially render the service “deaf” while one request is being handled – which may be for a very long time if a client is slow to receive all the data it has requested. Here a threading or forking server is appropriate.

In some cases, it may be appropriate to process part of a request synchronously, but to finish processing in a forked child depending on the request data. This can be implemented by using a
synchronous server and doing an explicit fork in the request handler class `handle()` method.

Another approach to handling multiple simultaneous requests in an environment that supports neither threads nor `fork()` (or where these are too expensive or inappropriate for the service) is to maintain an explicit table of partially finished requests and to use `select()` to decide which request to work on next (or whether to handle a new incoming request). This is particularly important for stream services where each client can potentially be connected for a long time (if threads or subprocesses cannot be used). See `asyncore` for another way to manage this.
class SocketServer. BaseServer(server_address, RequestHandlerClass)

This is the superclass of all Server objects in the module. It defines the interface, given below, but does not implement most of the methods, which is done in subclasses. The two parameters are stored in the respective server_address and RequestHandlerClass attributes.

fileno()

Return an integer file descriptor for the socket on which the server is listening. This function is most commonly passed to select.select(), to allow monitoring multiple servers in the same process.

handle_request()

Process a single request. This function calls the following methods in order: get_request(), verify_request(), and process_request(). If the user-provided handle() method of the handler class raises an exception, the server’s handle_error() method will be called. If no request is received within timeout seconds, handle_timeout() will be called and handle_request() will return.

serve_forever(poll_interval=0.5)

Handle requests until an explicit shutdown() request. Poll for shutdown every poll_interval seconds. Ignores the timeout attribute. If you need to do periodic tasks, do them in another thread.

shutdown()
Tell the `serve_forever()` loop to stop and wait until it does.

*New in version 2.6.*

**server_close()**
Clean up the server. May be overridden.

*New in version 2.6.*

**address_family**
The family of protocols to which the server's socket belongs. Common examples are `socket.AF_INET` and `socket.AF_UNIX`.

**RequestHandlerClass**
The user-provided request handler class; an instance of this class is created for each request.

**server_address**
The address on which the server is listening. The format of addresses varies depending on the protocol family; see the documentation for the `socket` module for details. For Internet protocols, this is a tuple containing a string giving the address, and an integer port number: `('127.0.0.1', 80)`, for example.

**socket**
The socket object on which the server will listen for incoming requests.

The server classes support the following class variables:

**allow_reuse_address**
Whether the server will allow the reuse of an address. This defaults to `False`, and can be set in subclasses to change the policy.

**request_queue_size**
The size of the request queue. If it takes a long time to process a single request, any requests that arrive while the server is busy are placed into a queue, up to request_queue_size requests. Once the queue is full, further requests from clients will get a “Connection denied” error. The default value is usually 5, but this can be overridden by subclasses.

socket_type
The type of socket used by the server; socket.SOCK_STREAM and socket.SOCK_DGRAM are two common values.

timeout
Timeout duration, measured in seconds, or None if no timeout is desired. If handle_request() receives no incoming requests within the timeout period, the handle_timeout() method is called.

There are various server methods that can be overridden by subclasses of base server classes like TCPServer; these methods aren’t useful to external users of the server object.

finish_request()
Actually processes the request by instantiating RequestHandlerClass and calling its handle() method.

get_request()
Must accept a request from the socket, and return a 2-tuple containing the new socket object to be used to communicate with the client, and the client’s address.

handle_error(request, client_address)
This function is called if the handle() method of a RequestHandlerClass instance raises an exception. The
default action is to print the traceback to standard output and continue handling further requests.

**handle_timeout()**
This function is called when the `timeout` attribute has been set to a value other than `None` and the timeout period has passed with no requests being received. The default action for forking servers is to collect the status of any child processes that have exited, while in threading servers this method does nothing.

**process_request**(request, client_address)
Calls `finish_request()` to create an instance of the `RequestHandlerClass`. If desired, this function can create a new process or thread to handle the request; the `ForkingMixIn` and `ThreadingMixIn` classes do this.

**server_activate()**
Called by the server's constructor to activate the server. The default behavior for a TCP server just invokes `listen()` on the server's socket. May be overridden.

**server_bind()**
Called by the server's constructor to bind the socket to the desired address. May be overridden.

**verify_request**(request, client_address)
Must return a Boolean value; if the value is `True`, the request will be processed, and if it's `False`, the request will be denied. This function can be overridden to implement access controls for a server. The default implementation always returns `True`. 
20.17.3. Request Handler Objects

class SocketServer.BaseRequestHandler

This is the superclass of all request handler objects. It defines the interface, given below. A concrete request handler subclass must define a new handle() method, and can override any of the other methods. A new instance of the subclass is created for each request.

setup()

Called before the handle() method to perform any initialization actions required. The default implementation does nothing.

handle()

This function must do all the work required to service a request. The default implementation does nothing. Several instance attributes are available to it; the request is available as self.request; the client address as self.client_address; and the server instance as self.server, in case it needs access to per-server information.

The type of self.request is different for datagram or stream services. For stream services, self.request is a socket object; for datagram services, self.request is a pair of string and socket.

finish()

Called after the handle() method to perform any clean-up actions required. The default implementation does nothing. If setup() raises an exception, this function will not be called.

class SocketServer.StreamRequestHandler
class SocketServer.DatagramRequestHandler

These BaseRequestHandler subclasses override the setup() and finish() methods, and provide self.rfile and self.wfile attributes. The self.rfile and self.wfile attributes can be read or written, respectively, to get the request data or return data to the client.
20.17.4. Examples

20.17.4.1. SocketServer.TCPServer Example

This is the server side:

```python
import SocketServer
class MyTCPHandler(SocketServer.BaseRequestHandler):
    """
    The request handler class for our server.
    It is instantiated once per connection to the server, and m
    override the handle() method to implement communication to
    client.
    """
    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print "{} wrote:".format(self.client_address[0])
        print self.data
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

if __name__ == '__main__':
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    server = SocketServer.TCPServer((HOST, PORT), MyTCPHandler)

    # Activate the server; this will keep running until you
    # interrupt the program with Ctrl-C
    server.serve_forever()
```

An alternative request handler class that makes use of streams (file-like objects that simplify communication by providing the standard file interface):

```python
class MyTCPHandler(SocketServer.StreamRequestHandler):
```
def handle(self):
    # self.rfile is a file-like object created by the handler
    # we can now use e.g. readline() instead of raw recv()
    self.data = self.rfile.readline().strip()
    print "{} wrote:".format(self.client_address[0])
    print self.data
    # Likewise, self.wfile is a file-like object used to write
    # to the client
    self.wfile.write(self.data.upper())

The difference is that the readline() call in the second handler will call recv() multiple times until it encounters a newline character, while the single recv() call in the first handler will just return what has been sent from the client in one sendall() call.

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(data + "\n")

    # Receive data from the server and shut down
    received = sock.recv(1024)
finally:
    sock.close()

print "Sent: {}".format(data)
print "Received: {}".format(received)
```

The output of the example should look something like this:
Server:

```
$ python TCPServer.py
127.0.0.1 wrote:
hello world with TCP
127.0.0.1 wrote:
python is nice
```

Client:

```
$ python TCPClient.py hello world with TCP
Sent: hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent: python is nice
Received: PYTHON IS NICE
```

20.17.4.2. SocketServer.UDPServer Example

This is the server side:

```python
import SocketServer

class MyUDPHandler(SocketServer.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and
    since there is no connection the client address must be given
    explicitly when sending data back via sendto().
    """

def handle(self):
    data = self.request[0].strip()
    socket = self.request[1]
    print "{} wrote:{}.format(self.client_address[0])
    print data
    socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    server = SocketServer.UDPServer((HOST, PORT), MyUDPHandler)
    server.serve_forever()
```
This is the client side:

```python
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connect.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(data + "\n", (HOST, PORT))
received = sock.recv(1024)

print "Sent: {}".format(data)
print "Received: {}".format(received)
```

The output of the example should look exactly like for the TCP server example.

### 20.17.4.3. Asynchronous Mixins

To build asynchronous handlers, use the `ThreadingMixIn` and `ForkingMixIn` classes.

An example for the `ThreadingMixIn` class:

```python
import socket
import threading
import SocketServer

class ThreadedTCPRequestHandler(SocketServer.BaseRequestHandler):
    def handle(self):
        data = self.request.recv(1024)
        cur_thread = threading.current_thread()
        response = "{}: {}".format(cur_thread.name, data)
        self.request.sendall(response)
```
```python
class ThreadedTCPServer(SocketServer.ThreadingMixIn, SocketServer.ThreadingTCPServer):
    pass

def client(ip, port, message):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((ip, port))
    try:
        sock.sendall(message)
        response = sock.recv(1024)
        print "Received: {}".format(response)
    finally:
        sock.close()

if __name__ == "__main__":
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPServer)
    ip, port = server.server_address

    # Start a thread with the server -- that thread will then start one more thread for each request
    server_thread = threading.Thread(target=server.serve_forever)
    server_thread.daemon = True
    server_thread.start()
    print "Server loop running in thread: ", server_thread.name

    client(ip, port, "Hello World 1")
    client(ip, port, "Hello World 2")
    client(ip, port, "Hello World 3")

    server.shutdown()
    server.server_close()
```

The output of the example should look something like this:

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

The **ForkingMixIn** class is used in the same way, except that the
server will spawn a new process for each request. Available only on POSIX platforms that support \texttt{fork()}. 
20.18. BaseHTTPServer — Basic HTTP server

Note: The BaseHTTPServer module has been merged into http.server in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/BaseHTTPServer.py

This module defines two classes for implementing HTTP servers (Web servers). Usually, this module isn’t used directly, but is used as a basis for building functioning Web servers. See the SimpleHTTPServer and CGIHTTPServer modules.

The first class, HTTPServer, is a SocketServer.TCPServer subclass, and therefore implements the SocketServer.BaseServer interface. It creates and listens at the HTTP socket, dispatching the requests to a handler. Code to create and run the server looks like this:

```python
def run(server_class=BaseHTTPServer.HTTPServer, 
        handler_class=BaseHTTPServer.BaseHTTPRequestHandler): 
    server_address = ('', 8000) 
    httpd = server_class(server_address, handler_class) 
    httpd.serve_forever()
```

class BaseHTTPServer.HTTPServer(server_address, RequestHandlerClass)

This class builds on the TCPServer class by storing the server address as instance variables named server_name and server_port. The server is accessible by the handler, typically through the handler’s server instance variable.

class BaseHTTPServer.BaseHTTPRequestHandler(request,
This class is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). `BaseHTTPRequestHandler` provides a number of class and instance variables, and methods for use by subclasses.

The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method `SPAM`, the `do_SPAM()` method will be called with no arguments. All of the relevant information is stored in instance variables of the handler. Subclasses should not need to override or extend the `__init__()` method.

`BaseHTTPRequestHandler` has the following instance variables:

- **client_address**
  Contains a tuple of the form `(host, port)` referring to the client's address.

- **server**
  Contains the server instance.

- **command**
  Contains the command (request type). For example, `'GET'`.

- **path**
  Contains the request path.

- **request_version**
  Contains the version string from the request. For example, `'HTTP/1.0'`.

- **headers**
  Holds an instance of the class specified by the `MessageClass`
class variable. This instance parses and manages the headers in the HTTP request.

rfile
Contains an input stream, positioned at the start of the optional input data.

wfile
Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream.

BaseHTTPRequestHandler has the following class variables:

server_version
Specifies the server software version. You may want to override this. The format is multiple whitespace-separated strings, where each string is of the form name[/version]. For example, 'BaseHTTP/0.2'.

sys_version
Contains the Python system version, in a form usable by the version_string method and the server_version class variable. For example, 'Python/1.4'.

ter_message_format
Specifies a format string for building an error response to the client. It uses parenthesized, keyed format specifiers, so the format operand must be a dictionary. The code key should be an integer, specifying the numeric HTTP error code value. message should be a string containing a (detailed) error message of what occurred, and explain should be an explanation of the error code number. Default message and explain values can found in the responses class variable.

ter_content_type
Specifies the Content-Type HTTP header of error responses sent to the client. The default value is 'text/html'.

*New in version 2.6:* Previously, the content type was always 'text/html'.

**protocol_version**
This specifies the HTTP protocol version used in responses. If set to 'HTTP/1.1', the server will permit HTTP persistent connections; however, your server must then include an accurate Content-Length header (using `send_header()`) in all of its responses to clients. For backwards compatibility, the setting defaults to 'HTTP/1.0'.

**MessageClass**
Specifies a `rfc822.Message`-like class to parse HTTP headers. Typically, this is not overridden, and it defaults to `mimetypes.Message`.

**responses**
This variable contains a mapping of error code integers to two-element tuples containing a short and long message. For example, `{code: (shortmessage, longmessage)}`. The shortmessage is usually used as the message key in an error response, and longmessage as the explain key (see the error_message_format class variable).

A `BaseHTTPRequestHandler` instance has the following methods:

**handle()**
Calls `handle_one_request()` once (or, if persistent connections are enabled, multiple times) to handle incoming HTTP requests. You should never need to override it; instead, implement appropriate `do_*( )` methods.
handle_one_request()
This method will parse and dispatch the request to the appropriate do_*() method. You should never need to override it.

send_error(code[, message])
Sends and logs a complete error reply to the client. The numeric code specifies the HTTP error code, with message as optional, more specific text. A complete set of headers is sent, followed by text composed using the error_message_format class variable. The body will be empty if the method is HEAD or the response code is one of the following: 1xx, 204 No Content, 205 Reset Content, 304 Not Modified.

send_response(code[, message])
Sends a response header and logs the accepted request. The HTTP response line is sent, followed by Server and Date headers. The values for these two headers are picked up from the version_string() and date_time_string() methods, respectively.

send_header(keyword, value)
Writes a specific HTTP header to the output stream. keyword should specify the header keyword, with value specifying its value.

end_headers()
Sends a blank line, indicating the end of the HTTP headers in the response.

log_request([code[, size]])
Logs an accepted (successful) request. code should specify
the numeric HTTP code associated with the response. If a size of the response is available, then it should be passed as the size parameter.

`log_error(...)`
Logs an error when a request cannot be fulfilled. By default, it passes the message to `log_message()`, so it takes the same arguments (format and additional values).

`log_message(format, ...)`
Logs an arbitrary message to `sys.stderr`. This is typically overridden to create custom error logging mechanisms. The format argument is a standard printf-style format string, where the additional arguments to `log_message()` are applied as inputs to the formatting. The client ip address and current date and time are prefixed to every message logged.

`version_string()`
Returns the server software's version string. This is a combination of the `server_version` and `sys_version` class variables.

`date_time_string([timestamp])`
Returns the date and time given by `timestamp` (which must be in the format returned by `time.time()`), formatted for a message header. If `timestamp` is omitted, it uses the current date and time.

The result looks like 'Sun, 06 Nov 1994 08:49:37 GMT'.

*New in version 2.5:* The `timestamp` parameter.

`log_date_time_string()`
Returns the current date and time, formatted for logging.
address_string()

Returns the client address, formatted for logging. A name lookup is performed on the client's IP address.
To create a server that doesn’t run forever, but until some condition is fulfilled:

```python
def run_while_true(server_class=BaseHTTPServer.HTTPServer,
                     handler_class=BaseHTTPServer.BaseHTTPRequestHandler):
    """
    This assumes that keep_running() is a function of no arguments
    that is tested initially and (re)after each request. If its return
    value is true, the server continues.
    """
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    while keep_running():
        httpd.handle_request()
```

See also:

**Module CGIHTTPServer**

Extended request handler that supports CGI scripts.

**Module SimpleHTTPServer**

Basic request handler that limits response to files actually under the document root.
20.19. SimpleHTTPServer — Simple HTTP request handler

**Note:** The SimpleHTTPServer module has been merged into http.server in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

The SimpleHTTPServer module defines a single class, SimpleHTTPRequestHandler, which is interface-compatible with BaseHTTPServer.BaseHTTPRequestHandler.

The SimpleHTTPServer module defines the following class:

```python
class SimpleHTTPServer.SimpleHTTPRequestHandler(request, client_address, server)
```

This class serves files from the current directory and below, directly mapping the directory structure to HTTP requests.

A lot of the work, such as parsing the request, is done by the base class BaseHTTPServer.BaseHTTPRequestHandler. This class implements the `do_GET()` and `do_HEAD()` functions.

The following are defined as class-level attributes of SimpleHTTPRequestHandler:

- **server_version**
  
  This will be "SimpleHTTP/" + __version__, where __version__ is defined at the module level.

- **extensions_map**
  
  A dictionary mapping suffixes into MIME types. The default is signified by an empty string, and is considered to be
The mapping is used case-insensitively, and so should contain only lower-cased keys.

The SimpleHTTPRequestHandler class defines the following methods:

**do_HEAD()**
This method serves the 'HEAD' request type: it sends the headers it would send for the equivalent GET request. See the do_GET() method for a more complete explanation of the possible headers.

**do_GET()**
The request is mapped to a local file by interpreting the request as a path relative to the current working directory.

If the request was mapped to a directory, the directory is checked for a file named index.html or index.htm (in that order). If found, the file's contents are returned; otherwise a directory listing is generated by calling the list_directory() method. This method uses os.listdir() to scan the directory, and returns a 404 error response if the listdir() fails.

If the request was mapped to a file, it is opened and the contents are returned. Any IOError exception in opening the requested file is mapped to a 404, 'File not found' error. Otherwise, the content type is guessed by calling the guess_type() method, which in turn uses the extensions_map variable.

A 'Content-type:' header with the guessed content type is output, followed by a 'Content-Length:' header with the file's size and a 'Last-Modified:' header with the file's modification time.
Then follows a blank line signifying the end of the headers, and then the contents of the file are output. If the file's MIME type starts with text/ the file is opened in text mode; otherwise binary mode is used.

The test() function in the SimpleHTTPServer module is an example which creates a server using the SimpleHTTPRequestHandler as the Handler.

*New in version 2.5:* The 'Last-Modified' header.

The SimpleHTTPServer module can be used in the following manner in order to set up a very basic web server serving files relative to the current directory.

```python
import SimpleHTTPServer
import SocketServer

PORT = 8000

Handler = SimpleHTTPServer.SimpleHTTPRequestHandler

httpd = SocketServer.TCPServer(('', PORT), Handler)

print "serving at port", PORT

httpd.serve_forever()
```

The SimpleHTTPServer module can also be invoked directly using the -m switch of the interpreter with a port number argument. Similar to the previous example, this serves the files relative to the current directory.

```bash
python -m SimpleHTTPServer 8000
```

**See also:**

**Module** BaseHTTPServer
Base class implementation for Web server and request handler.
Note: The CGIHTTPServer module has been merged into http.server in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

The CGIHTTPServer module defines a request-handler class, interface compatible with `BaseHTTPServer.BaseHTTPRequestHandler` and inherits behavior from `SimpleHTTPServer.SimpleHTTPRequestHandler` but can also run CGI scripts.

Note: This module can run CGI scripts on Unix and Windows systems.

Note: CGI scripts run by the CGIHTTPRequestHandler class cannot execute redirects (HTTP code 302), because code 200 (script output follows) is sent prior to execution of the CGI script. This pre-empts the status code.

The CGIHTTPServer module defines the following class:

```python
class CGIHTTPServer.CGIHTTPRequestHandler(request, client_address, server)
```

This class is used to serve either files or output of CGI scripts from the current directory and below. Note that mapping HTTP hierarchic structure to local directory structure is exactly as in `SimpleHTTPServer.SimpleHTTPRequestHandler`.

The class will however, run the CGI script, instead of serving it as a file, if it guesses it to be a CGI script. Only directory-based CGI are used — the other common server configuration is to treat
special extensions as denoting CGI scripts.

The `do_GET()` and `do_HEAD()` functions are modified to run CGI scripts and serve the output, instead of serving files, if the request leads to somewhere below the `cgi_directories` path.

The `CGIHTTPRequestHandler` defines the following data member:

`cgi_directories`

This defaults to `['/cgi-bin', '/htbin']` and describes directories to treat as containing CGI scripts.

The `CGIHTTPRequestHandler` defines the following methods:

`do_POST()`

This method serves the `POST` request type, only allowed for CGI scripts. Error 501, “Can only POST to CGI scripts”, is output when trying to POST to a non-CGI url.

Note that CGI scripts will be run with UID of user nobody, for security reasons. Problems with the CGI script will be translated to error 403.

For example usage, see the implementation of the `test()` function.

**See also:**

**Module** `BaseHTTPServer`

Base class implementation for Web server and request handler.
20.21. **cookielib** — Cookie handling for HTTP clients

**Note:** The `cookielib` module has been renamed to `http.cookiejar` in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

*New in version 2.4.*

**Source code:** [Lib/cookielib.py](Lib/cookielib.py)

The `cookielib` module defines classes for automatic handling of HTTP cookies. It is useful for accessing web sites that require small pieces of data – *cookies* – to be set on the client machine by an HTTP response from a web server, and then returned to the server in later HTTP requests.

Both the regular Netscape cookie protocol and the protocol defined by [RFC 2965](https://tools.ietf.org/html/rfc2965) are handled. RFC 2965 handling is switched off by default. [RFC 2109](https://tools.ietf.org/html/rfc2109) cookies are parsed as Netscape cookies and subsequently treated either as Netscape or RFC 2965 cookies according to the ‘policy’ in effect. Note that the great majority of cookies on the Internet are Netscape cookies. `cookielib` attempts to follow the de-facto Netscape cookie protocol (which differs substantially from that set out in the original Netscape specification), including taking note of the `max-age` and `port` cookie-attributes introduced with RFC 2965.

**Note:** The various named parameters found in *Set-Cookie* and *Set-Cookie2* headers (eg. `domain` and `expires`) are conventionally referred to as *attributes*. To distinguish them from Python
attributes, the documentation for this module uses the term *cookie-attribute* instead.

The module defines the following exception:

```python
exception cookielib.LoadError
```

Instances of `FileCookieJar` raise this exception on failure to load cookies from a file.

**Note:** For backwards-compatibility with Python 2.4 (which raised an `IOError`), `LoadError` is a subclass of `IOError`.

The following classes are provided:

```python
class cookielib.CookieJar(policy=None)
```

`policy` is an object implementing the `CookiePolicy` interface.

The `CookieJar` class stores HTTP cookies. It extracts cookies from HTTP requests, and returns them in HTTP responses. `CookieJar` instances automatically expire contained cookies when necessary. Subclasses are also responsible for storing and retrieving cookies from a file or database.

```python
class cookielib.FileCookieJar(filename, delayload=None, policy=None)
```

`policy` is an object implementing the `CookiePolicy` interface. For the other arguments, see the documentation for the corresponding attributes.

A `CookieJar` which can load cookies from, and perhaps save cookies to, a file on disk. Cookies are NOT loaded from the named file until either the `load()` or `revert()` method is called. Subclasses of this class are documented in section `FileCookieJar subclasses and co-operation with web browsers`. 


class cookielib.CookiePolicy

This class is responsible for deciding whether each cookie should be accepted from / returned to the server.

class cookielib.DefaultCookiePolicy(blocked_domains=None, allowed_domains=None, netscape=True, rfc2965=False, rfc2109_as_netscape=None, hide_cookie2=False, strict_domain=False, strict_rfc2965_unverifiable=True, strict_ns_unverifiable=False, strict_ns_domain=DefaultCookiePolicy.DomainLiberal, strict_ns_set_initial_dollar=False, strict_ns_set_path=False)

Constructor arguments should be passed as keyword arguments only. blocked_domains is a sequence of domain names that we never accept cookies from, nor return cookies to. allowed_domains if not None, this is a sequence of the only domains for which we accept and return cookies. For all other arguments, see the documentation for CookiePolicy and DefaultCookiePolicy objects.

DefaultCookiePolicy implements the standard accept / reject rules for Netscape and RFC 2965 cookies. By default, RFC 2109 cookies (ie. cookies received in a Set-Cookie header with a version cookie-attribute of 1) are treated according to the RFC 2965 rules. However, if RFC 2965 handling is turned off or rfc2109_as_netscape is True, RFC 2109 cookies are ‘downgraded’ by the CookieJar instance to Netscape cookies, by setting the version attribute of the Cookie instance to 0. DefaultCookiePolicy also provides some parameters to allow some fine-tuning of policy.

class cookielib.Cookie

This class represents Netscape, RFC 2109 and RFC 2965 cookies. It is not expected that users of cookielib construct their own Cookie instances. Instead, if necessary, call make_cookies()
on a `CookieJar` instance.

**See also:**

**Module urllib2**  
URL opening with automatic cookie handling.

**Module Cookie**  
HTTP cookie classes, principally useful for server-side code. The `cookielib` and `Cookie` modules do not depend on each other.

[https://curl.haxx.se/rfc/cookie_spec.html](https://curl.haxx.se/rfc/cookie_spec.html)  
The specification of the original Netscape cookie protocol. Though this is still the dominant protocol, the ‘Netscape cookie protocol’ implemented by all the major browsers (and `cookielib`) only bears a passing resemblance to the one sketched out in `cookie_spec.html`.

**RFC 2109 - HTTP State Management Mechanism**  
Obsoleted by RFC 2965. Uses `Set-Cookie` with version=1.

**RFC 2965 - HTTP State Management Mechanism**  
The Netscape protocol with the bugs fixed. Uses `Set-Cookie2` in place of `Set-Cookie`. Not widely used.

[http://kristol.org/cookie/errata.html](http://kristol.org/cookie/errata.html)  
Unfinished errata to RFC 2965.

**RFC 2964 - Use of HTTP State Management**

CookieJar objects support the iterator protocol for iterating over contained Cookie objects.

CookieJar has the following methods:

CookieJar.add_cookie_header(request)
Add correct Cookie header to request.

If policy allows (ie. the rfc2965 and hide_cookie2 attributes of the CookieJar’s CookiePolicy instance are true and false respectively), the Cookie2 header is also added when appropriate.

The request object (usually a urllib2.Request instance) must support the methods get_full_url(), get_host(), get_type(), unverifiable(), get_origin_req_host(), has_header(), get_header(), header_items(), and add_unredirected_header(), as documented by urllib2.

CookieJar.extract_cookies(response, request)
Extract cookies from HTTP response and store them in the CookieJar, where allowed by policy.

The CookieJar will look for allowable Set-Cookie and Set-Cookie2 headers in the response argument, and store cookies as appropriate (subject to the CookiePolicy.set_ok() method’s approval).

The response object (usually the result of a call to
urllib2.urlopen(), or similar) should support an info() method, which
returns an object with a getallmatchingheaders() method
(usually a mimetools.Message instance).

The request object (usually a urllib2.Request instance) must
support the methods get_full_url(), get_host(),
unverifiable(), and get_origin_req_host(), as documented by
urllib2. The request is used to set default values for cookie-
attributes as well as for checking that the cookie is allowed to be
set.

CookieJar.set_policy(policy)
Set the CookiePolicy instance to be used.

CookieJar.make_cookies(response, request)
Return sequence of Cookie objects extracted from response
object.

See the documentation for extract_cookies() for the interfaces
required of the response and request arguments.

CookieJar.set_cookie_if_ok(cookie, request)
Set a Cookie if policy says it's OK to do so.

CookieJar.set_cookie(cookie)
Set a Cookie, without checking with policy to see whether or not it
should be set.

CookieJar.clear([domain[, path[, name]]])
Clear some cookies.

If invoked without arguments, clear all cookies. If given a single
argument, only cookies belonging to that domain will be removed.
If given two arguments, cookies belonging to the specified
domain and URL path are removed. If given three arguments, then the cookie with the specified domain, path and name is removed.

Raises KeyError if no matching cookie exists.

CookieJar.clear_session_cookies()
Discard all session cookies.

Discards all contained cookies that have a true discard attribute (usually because they had either no max-age or expires cookie-attribute, or an explicit discard cookie-attribute). For interactive browsers, the end of a session usually corresponds to closing the browser window.

Note that the save() method won’t save session cookies anyway, unless you ask otherwise by passing a true ignore_discard argument.

FileCookieJar implements the following additional methods:

FileCookieJar.save(filename=None, ignore_discard=False, ignore_expires=False)
Save cookies to a file.

This base class raises NotImplementedError. Subclasses may leave this method unimplemented.

filename is the name of file in which to save cookies. If filename is not specified, self.filename is used (whose default is the value passed to the constructor, if any); if self.filename is None, ValueError is raised.

ignore_discard: save even cookies set to be discarded.
ignore_expires: save even cookies that have expired
The file is overwritten if it already exists, thus wiping all the cookies it contains. Saved cookies can be restored later using the `load()` or `revert()` methods.

FileCookieJar. **load**(*filename=None, ignore_discard=False, ignore_expires=False*)

Load cookies from a file.

Old cookies are kept unless overwritten by newly loaded ones.

Arguments are as for `save()`.

The named file must be in the format understood by the class, or `LoadError` will be raised. Also, `IOError` may be raised, for example if the file does not exist.

**Note:** For backwards-compatibility with Python 2.4 (which raised an `IOError`), `LoadError` is a subclass of `IOError`.

FileCookieJar. **revert**(*filename=None, ignore_discard=False, ignore_expires=False*)

Clear all cookies and reload cookies from a saved file.

`revert()` can raise the same exceptions as `load()`. If there is a failure, the object’s state will not be altered.

FileCookieJar instances have the following public attributes:

FileCookieJar. **filename**

Filename of default file in which to keep cookies. This attribute may be assigned to.

FileCookieJar. **delayload**

If true, load cookies lazily from disk. This attribute should not be assigned to. This is only a hint, since this only affects
performance, not behaviour (unless the cookies on disk are changing). A \texttt{CookieJar} object may ignore it. None of the \texttt{FileCookieJar} classes included in the standard library lazily loads cookies.
20.21.2. FileCookieJar subclasses and co-operation with web browsers

The following CookieJar subclasses are provided for reading and writing.

```
class cookielib.MozillaCookieJar(filename, delayload=None, policy=None)
```

A FileCookieJar that can load from and save cookies to disk in the Mozilla cookies.txt file format (which is also used by the Lynx and Netscape browsers).

**Note:** Version 3 of the Firefox web browser no longer writes cookies in the cookies.txt file format.

**Note:** This loses information about RFC 2965 cookies, and also about newer or non-standard cookie-attributes such as port.

**Warning:** Back up your cookies before saving if you have cookies whose loss / corruption would be inconvenient (there are some subtleties which may lead to slight changes in the file over a load / save round-trip).

Also note that cookies saved while Mozilla is running will get clobbered by Mozilla.

```
class cookielib.LWPCookieJar(filename, delayload=None, policy=None)
```

A FileCookieJar that can load from and save cookies to disk in format compatible with the libwww-perl library’s Set-Cookie3 file format. This is convenient if you want to store cookies in a
human-readable file.
20.21.3. CookiePolicy Objects

Objects implementing the CookiePolicy interface have the following methods:

CookiePolicy.set_ok(cookie, request)
Return boolean value indicating whether cookie should be accepted from server.

cookie is a cookielib.Cookie instance. request is an object implementing the interface defined by the documentation for CookieJar.extract_cookies().

CookiePolicy.return_ok(cookie, request)
Return boolean value indicating whether cookie should be returned to server.

cookie is a cookielib.Cookie instance. request is an object implementing the interface defined by the documentation for CookieJar.add_cookie_header().

CookiePolicy.domain_return_ok(domain, request)
Return false if cookies should not be returned, given cookie domain.

This method is an optimization. It removes the need for checking every cookie with a particular domain (which might involve reading many files). Returning true from domain_return_ok() and path_return_ok() leaves all the work to return_ok().

If domain_return_ok() returns true for the cookie domain, path_return_ok() is called for the cookie path. Otherwise, path_return_ok() and return_ok() are never called for that
cookie domain. If `path_return_ok()` returns true, `return_ok()` is called with the `Cookie` object itself for a full check. Otherwise, `return_ok()` is never called for that cookie path.

Note that `domain_return_ok()` is called for every cookie domain, not just for the request domain. For example, the function might be called with both `.example.com` and `www.example.com` if the request domain is `www.example.com`. The same goes for `path_return_ok()`.

The `request` argument is as documented for `return_ok()`.

CookiePolicy. `path_return_ok(path, request)`

Return false if cookies should not be returned, given cookie path.

See the documentation for `domain_return_ok()`.

In addition to implementing the methods above, implementations of the `CookiePolicy` interface must also supply the following attributes, indicating which protocols should be used, and how. All of these attributes may be assigned to.

CookiePolicy. `netscape`

Implement Netscape protocol.

CookiePolicy. `rfc2965`

Implement RFC 2965 protocol.

CookiePolicy. `hide_cookie2`

Don’t add `Cookie2` header to requests (the presence of this header indicates to the server that we understand RFC 2965 cookies).

The most useful way to define a `CookiePolicy` class is by subclassing from `DefaultCookiePolicy` and overriding some or all of
the methods above. CookiePolicy itself may be used as a ‘null policy’ to allow setting and receiving any and all cookies (this is unlikely to be useful).
20.21.4. DefaultCookiePolicy Objects

Implements the standard rules for accepting and returning cookies.

Both RFC 2965 and Netscape cookies are covered. RFC 2965 handling is switched off by default.

The easiest way to provide your own policy is to override this class and call its methods in your overridden implementations before adding your own additional checks:

```python
import cookielib
class MyCookiePolicy(cookielib.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not cookielib.DefaultCookiePolicy.set_ok(self, cookie):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

In addition to the features required to implement the `CookiePolicy` interface, this class allows you to block and allow domains from setting and receiving cookies. There are also some strictness switches that allow you to tighten up the rather loose Netscape protocol rules a little bit (at the cost of blocking some benign cookies).

A domain blacklist and whitelist is provided (both off by default). Only domains not in the blacklist and present in the whitelist (if the whitelist is active) participate in cookie setting and returning. Use the `blocked_domains` constructor argument, and `blocked_domains()` and `set_blocked_domains()` methods (and the corresponding argument and methods for `allowed_domains`). If you set a whitelist, you can turn it off again by setting it to `None`. 
Domains in block or allow lists that do not start with a dot must equal the cookie domain to be matched. For example, "example.com" matches a blacklist entry of "example.com", but "www.example.com" does not. Domains that do start with a dot are matched by more specific domains too. For example, both "www.example.com" and "www.coyote.example.com" match ".example.com" (but "example.com" itself does not). IP addresses are an exception, and must match exactly. For example, if blocked_domains contains "192.168.1.2" and ".168.1.2", 192.168.1.2 is blocked, but 193.168.1.2 is not.

DefaultCookiePolicy implements the following additional methods:

DefaultCookiePolicy.**blocked_domains**()

Return the sequence of blocked domains (as a tuple).

DefaultCookiePolicy.**set_blocked_domains**(blocked_domains)

Set the sequence of blocked domains.

DefaultCookiePolicy.**is_blocked**(domain)

Return whether domain is on the blacklist for setting or receiving cookies.

DefaultCookiePolicy.**allowed_domains**()

Return None, or the sequence of allowed domains (as a tuple).

DefaultCookiePolicy.**set_allowed_domains**(allowed_domains)

Set the sequence of allowed domains, or None.

DefaultCookiePolicy.**is_not_allowed**(domain)

Return whether domain is not on the whitelist for setting or receiving cookies.

DefaultCookiePolicy instances have the following attributes, which are all initialised from the constructor arguments of the same name, and which may all be assigned to.
DefaultCookiePolicy. **rfc2109_as_netscape**

If true, request that the `CookieJar` instance downgrade RFC 2109 cookies (ie. cookies received in a `Set-Cookie` header with a version cookie-attribute of 1) to Netscape cookies by setting the version attribute of the `Cookie` instance to 0. The default value is `None`, in which case RFC 2109 cookies are downgraded if and only if RFC 2965 handling is turned off. Therefore, RFC 2109 cookies are downgraded by default.

*New in version 2.5.*

General strictness switches:

DefaultCookiePolicy. **strict_domain**

Don’t allow sites to set two-component domains with country-code top-level domains like `.co.uk`, `.gov.uk`, `.co.nz` etc. This is far from perfect and isn’t guaranteed to work!

RFC 2965 protocol strictness switches:

DefaultCookiePolicy. **strict_rfc2965_unverifiable**

Follow RFC 2965 rules on unverifiable transactions (usually, an unverifiable transaction is one resulting from a redirect or a request for an image hosted on another site). If this is false, cookies are *never* blocked on the basis of verifiability

Netscape protocol strictness switches:

DefaultCookiePolicy. **strict_ns_unverifiable**

Apply RFC 2965 rules on unverifiable transactions even to Netscape cookies.

DefaultCookiePolicy. **strict_ns_domain**

Flags indicating how strict to be with domain-matching rules for Netscape cookies. See below for acceptable values.
DefaultCookiePolicy. **strict_ns_set_initial_dollar**

Ignore cookies in Set-Cookie: headers that have names starting with '$'.

DefaultCookiePolicy. **strict_ns_set_path**

Don’t allow setting cookies whose path doesn’t path-match request URI.

**strict_ns_domain** is a collection of flags. Its value is constructed by or-ing together (for example, DomainStrictNoDots|DomainStrictNonDomain means both flags are set).

DefaultCookiePolicy. **DomainStrictNoDots**

When setting cookies, the ‘host prefix’ must not contain a dot (eg. www.foo.bar.com can’t set a cookie for .bar.com, because www.foo contains a dot).

DefaultCookiePolicy. **DomainStrictNonDomain**

Cookies that did not explicitly specify a domain cookie-attribute can only be returned to a domain equal to the domain that set the cookie (eg. spam.example.com won’t be returned cookies from example.com that had no domain cookie-attribute).

DefaultCookiePolicy. **DomainRFC2965Match**

When setting cookies, require a full RFC 2965 domain-match.

The following attributes are provided for convenience, and are the most useful combinations of the above flags:

DefaultCookiePolicy. **DomainLiberal**

Equivalent to 0 (ie. all of the above Netscape domain strictness flags switched off).

DefaultCookiePolicy. **DomainStrict**
Equivalent to DomainStrictNoDots|DomainStrictNonDomain.
20.21.5. Cookie Objects

Cookie instances have Python attributes roughly corresponding to the standard cookie-attributes specified in the various cookie standards. The correspondence is not one-to-one, because there are complicated rules for assigning default values, because the max-age and expires cookie-attributes contain equivalent information, and because RFC 2109 cookies may be ‘downgraded’ by cookielib from version 1 to version 0 (Netscape) cookies.

Assignment to these attributes should not be necessary other than in rare circumstances in a CookiePolicy method. The class does not enforce internal consistency, so you should know what you’re doing if you do that.

Cookie. version
Integer or None. Netscape cookies have version 0. RFC 2965 and RFC 2109 cookies have a version cookie-attribute of 1. However, note that cookielib may ‘downgrade’ RFC 2109 cookies to Netscape cookies, in which case version is 0.

Cookie. name
Cookie name (a string).

Cookie. value
Cookie value (a string), or None.

Cookie. port
String representing a port or a set of ports (eg. ‘80’, or ‘80,8080’), or None.

Cookie. path
Cookie path (a string, eg. '/acme/rocket_launchers').
Cookie. `secure`  
   `True` if cookie should only be returned over a secure connection.

Cookie. `expires`  
   Integer expiry date in seconds since epoch, or `None`. See also the `is_expired()` method.

Cookie. `discard`  
   `True` if this is a session cookie.

Cookie. `comment`  
   String comment from the server explaining the function of this cookie, or `None`.

Cookie. `comment_url`  
   URL linking to a comment from the server explaining the function of this cookie, or `None`.

Cookie. `rfc2109`  
   `True` if this cookie was received as an RFC 2109 cookie (ie. the cookie arrived in a `Set-Cookie` header, and the value of the Version cookie-attribute in that header was 1). This attribute is provided because `cookielib` may ‘downgrade’ RFC 2109 cookies to Netscape cookies, in which case `version` is 0.

   *New in version 2.5.*

Cookie. `port_specified`  
   `True` if a port or set of ports was explicitly specified by the server (in the `Set-Cookie` / `Set-Cookie2` header).

Cookie. `domain_specified`  
   `True` if a domain was explicitly specified by the server.

Cookie. `domain_initial_dot`  
   `True` if the domain explicitly specified by the server began with a
Cookies may have additional non-standard cookie-attributes. These may be accessed using the following methods:

Cookie.**has_nonstandard_attr**(name)

Return true if cookie has the named cookie-attribute.

Cookie.**get_nonstandard_attr**(name, default=None)

If cookie has the named cookie-attribute, return its value. Otherwise, return default.

Cookie.**set_nonstandard_attr**(name, value)

Set the value of the named cookie-attribute.

The **Cookie** class also defines the following method:

Cookie.**is_expired**(now=None)

True if cookie has passed the time at which the server requested it should expire. If now is given (in seconds since the epoch), return whether the cookie has expired at the specified time.
20.21.6. Examples

The first example shows the most common usage of `cookielib`:

```python
import cookielib, urllib2
cj = cookielib.CookieJar()
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

This example illustrates how to open a URL using your Netscape, Mozilla, or Lynx cookies (assumes Unix/Netscape convention for location of the cookies file):

```python
import os, cookielib, urllib2
cj = cookielib.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "coo
cj = CookieJar(policy)
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

The next example illustrates the use of `DefaultCookiePolicy`. Turn on RFC 2965 cookies, be more strict about domains when setting and returning Netscape cookies, and block some domains from setting cookies or having them returned:

```python
import urllib2
from cookielib import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=DefaultCookiePolicy.DomainSt
blocked_domains=["ads.net", ".ads.net"]
)
20.22. **Cookie — HTTP state management**

**Note:** The `Cookie` module has been renamed to `http.cookies` in Python 3. The `2to3` tool will automatically adapt imports when converting your sources to Python 3.

**Source code:** Lib/Cookie.py

The `Cookie` module defines classes for abstracting the concept of cookies, an HTTP state management mechanism. It supports both simple string-only cookies, and provides an abstraction for having any serializable data-type as cookie value.

The module formerly strictly applied the parsing rules described in the [RFC 2109](https://tools.ietf.org/html/rfc2109) and [RFC 2068](https://tools.ietf.org/html/rfc2068) specifications. It has since been discovered that MSIE 3.0x doesn’t follow the character rules outlined in those specs and also many current day browsers and servers have relaxed parsing rules when comes to Cookie handling. As a result, the parsing rules used are a bit less strict.

The character set, `string.ascii_letters`, `string.digits` and `!#$%&'*+-.^_`~ denote the set of valid characters allowed by this module in Cookie name (as *key*).

**Note:** On encountering an invalid cookie, `CookieError` is raised, so if your cookie data comes from a browser you should always prepare for invalid data and catch `CookieError` on parsing.

```
exception Cookie.CookieError
    Exception failing because of RFC 2109 invalidity: incorrect
```
attributes, incorrect Set-Cookie header, etc.

```python
class Cookie.BaseCookie([input])
```
This class is a dictionary-like object whose keys are strings and whose values are `Morsel` instances. Note that upon setting a key to a value, the value is first converted to a `Morsel` containing the key and the value.

If `input` is given, it is passed to the `load()` method.

```python
class Cookie.SimpleCookie([input])
```
This class derives from `BaseCookie` and overrides `value_decode()` and `value_encode()` to be the identity and `str()` respectively.

```python
class Cookie.SerialCookie([input])
```
This class derives from `BaseCookie` and overrides `value_decode()` and `value_encode()` to be `pickle.loads()` and `pickle.dumps()`.

```
Deprecated since version 2.3: Reading pickled values from untrusted cookie data is a huge security hole, as pickle strings can be crafted to cause arbitrary code to execute on your server. It is supported for backwards compatibility only, and may eventually go away.
```

```python
class Cookie.SmartCookie([input])
```
This class derives from `BaseCookie`. It overrides `value_decode()` to be `pickle.loads()` if it is a valid pickle, and otherwise the value itself. It overrides `value_encode()` to be `pickle.dumps()` unless it is a string, in which case it returns the value itself.

```
Deprecated since version 2.3: The same security warning from
```
A further security note is warranted. For backwards compatibility, the `Cookie` module exports a class named `cookie` which is just an alias for `SmartCookie`. This is probably a mistake and will likely be removed in a future version. You should not use the `cookie` class in your applications, for the same reason why you should not use the `SerialCookie` class.

**See also:**

**Module** cookielib
HTTP cookie handling for web clients. The `cookielib` and `Cookie` modules do not depend on each other.

**RFC 2109 - HTTP State Management Mechanism**
This is the state management specification implemented by this module.
20.22.1. Cookie Objects

BaseCookie. **value_decode**(val)
Return a decoded value from a string representation. Return value can be any type. This method does nothing in BaseCookie — it exists so it can be overridden.

BaseCookie. **value_encode**(val)
Return an encoded value. val can be any type, but return value must be a string. This method does nothing in BaseCookie — it exists so it can be overridden.

In general, it should be the case that value_encode() and value_decode() are inverses on the range of value_decode.

BaseCookie. **output**(attrs[, header[, sep]])
Return a string representation suitable to be sent as HTTP headers. attrs and header are sent to each Morsel’s output() method. sep is used to join the headers together, and is by default the combination '\r\n' (CRLF).

*Changed in version 2.5:* The default separator has been changed from '\n' to match the cookie specification.

BaseCookie. js_output([[attrs]])
Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP headers was sent.

The meaning for attrs is the same as in output().

BaseCookie. **load**(rawdata)
If rawdata is a string, parse it as an HTTP_COOKIE and add the
values found there as **Morsels**. If it is a dictionary, it is equivalent to:

```python
for k, v in rawdata.items():
    cookie[k] = v
```
20.22.2. Morsel Objects

class Cookie.Morsel

Abstract a key/value pair, which has some RFC 2109 attributes.

Morsels are dictionary-like objects, whose set of keys is constant — the valid RFC 2109 attributes, which are

- expires
- path
- comment
- domain
- max-age
- secure
- version
- httponly

The attribute `httponly` specifies that the cookie is only transferred in HTTP requests, and is not accessible through JavaScript. This is intended to mitigate some forms of cross-site scripting.

The keys are case-insensitive.

_New in version 2.6:_ The `httponly` attribute was added.

Morsel. `value`

The value of the cookie.

Morsel. `coded_value`

The encoded value of the cookie — this is what should be sent.

Morsel. `key`

The name of the cookie.
Morsel. set(key, value, coded_value)

Set the key, value and coded_value attributes.

Morsel. isReservedKey(K)

Whether K is a member of the set of keys of a Morsel.

Morsel. output([attrs[, header]])

Return a string representation of the Morsel, suitable to be sent as an HTTP header. By default, all the attributes are included, unless attrs is given, in which case it should be a list of attributes to use. header is by default "Set-Cookie:"

Morsel. js_output([attrs])

Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP header was sent.

The meaning for attrs is the same as in output().

Morsel. OutputString([attrs])

Return a string representing the Morsel, without any surrounding HTTP or JavaScript.

The meaning for attrs is the same as in output().
The following example demonstrates how to use the `Cookie` module.

```python
>>> import Cookie
>>> C = Cookie.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print C  # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print C.output()  # same thing
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C = Cookie.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print C.output(header="Cookie:")
Cookie: rocky=road; Path=/cookie
>>> print C.output(attrs=[], header="Cookie:")
Cookie: rocky=road
>>> C = Cookie.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger")  # load from a string (HTTP header)
>>> print C
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = Cookie.SimpleCookie()
>>> C.load('keebler="E=everybody; L="Loves\"; fudge=\012;"')
>>> print C
Set-Cookie: keebler="E=everybody; L="Loves\"; fudge=\012;"
>>> C = Cookie.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/
>>> print C
Set-Cookie: oreo=doublestuff; Path=/
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = Cookie.SimpleCookie()
>>> C["number"] = 7  # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
```
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number=7
Set-Cookie: string=seven
>>> # SerialCookie and SmartCookie are deprecated
>>> # using it can cause security loopholes in your code.
>>> C = Cookie.SerialCookie()
>>> C["number"] = 7
>>> C["string"] = "seven"
>>> C["number"].value
7
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number="I7\012.
Set-Cookie: string="seven'\012p1\012."
>>> C = Cookie.SmartCookie()
>>> C["number"] = 7
>>> C["string"] = "seven"
>>> C["number"].value
7
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number="I7\012.
Set-Cookie: string=seven
20.23. xmlrpclib — XML-RPC client access

Note: The xmlrpclib module has been renamed to xmlrpc.client in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

New in version 2.2.

Source code: Lib/xmlrpclib.py

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP(S) as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

Warning: The xmlrpclib module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

Changed in version 2.7.9: For HTTPS URIs, xmlrpclib now performs all the necessary certificate and hostname checks by default.

class xmlrpclib.ServerProxy(uri[, transport[, encoding[, verbose[, allow_none[, use_datetime[, context]]]]]])

A ServerProxy instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL
of the server. The optional second argument is a transport factory instance; by default it is an internal SafeTransport instance for https: URLs and an internal HTTP Transport instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag.

The following parameters govern the use of the returned proxy instance. If allow_none is true, the Python constant None will be translated into XML; the default behaviour is for None to raise a TypeError. This is a commonly-used extension to the XML-RPC specification, but isn't supported by all clients and servers; see http://ontosys.com/xml-rpc/extensions.php for a description. The use_datetime flag can be used to cause date/time values to be presented as datetime.datetime objects; this is false by default. datetime.datetime objects may be passed to calls.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication: http://user:pass@host:port/path. The user:pass portion will be base64-encoded as an HTTP ‘Authorization’ header, and sent to the remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password. If an HTTPS URL is provided, context may be ssl.SSLContext and configures the SSL settings of the underlying HTTPS connection.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.
Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type):

<table>
<thead>
<tr>
<th>XML-RPC type</th>
<th>Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>bool</td>
</tr>
<tr>
<td>int Or i4</td>
<td>int or long in range from -2147483648 to 2147483647.</td>
</tr>
<tr>
<td>double</td>
<td>float</td>
</tr>
<tr>
<td>string</td>
<td>str Or unicode</td>
</tr>
<tr>
<td>array</td>
<td>list or tuple containing conformable elements. Arrays are returned as lists.</td>
</tr>
<tr>
<td>struct</td>
<td>dict. Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in; only their <strong>dict</strong> attribute is transmitted.</td>
</tr>
<tr>
<td>dateTime.iso8601</td>
<td>DateTime or datetime.datetime. Returned type depends on values of the use_datetime flags.</td>
</tr>
<tr>
<td>base64</td>
<td>Binary</td>
</tr>
<tr>
<td>nil</td>
<td>The None constant. Passing is allowed only if allow_none is true.</td>
</tr>
</tbody>
</table>

This is the full set of data types supported by XML-RPC. Method calls may also raise a special Fault instance, used to signal XML-RPC server errors, or ProtocolError used to signal an error in the HTTP/HTTPS transport layer. Both Fault and ProtocolError derive from a base class called Error. Note that even though starting with Python 2.2 you can subclass built-in types, the xmlrpclib module currently does not marshal instances of such subclasses.

When passing strings, characters special to XML such as <, >, and & will be automatically escaped. However, it’s the caller’s responsibility to ensure that the string is free of characters that
aren’t allowed in XML, such as the control characters with ASCII values between 0 and 31 (except, of course, tab, newline and carriage return); failing to do this will result in an XML-RPC request that isn’t well-formed XML. If you have to pass arbitrary strings via XML-RPC, use the `Binary` wrapper class described below.

`Server` is retained as an alias for `ServerProxy` for backwards compatibility. New code should use `ServerProxy`.

*Changed in version 2.5:* The `use_datetime` flag was added.

*Changed in version 2.6:* Instances of `new-style classes` can be passed in if they have an `__dict__` attribute and don’t have a base class that is marshalled in a special way.

*Changed in version 2.7.9:* Added the `context` argument.

**See also:**

**XML-RPC HOWTO**
A good description of XML-RPC operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.

**XML-RPC Introspection**
Describes the XML-RPC protocol extension for introspection.

**XML-RPC Specification**
The official specification.

**Unofficial XML-RPC Errata**
Fredrik Lundh’s “unofficial errata, intended to clarify certain details in the XML-RPC specification, as well as hint at ‘best practices’ to use when designing your own XML-RPC implementations.”
20.23.1. ServerProxy Objects

A `ServerProxy` instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a `Fault` or `ProtocolError` object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved `system` attribute:

```python
ServerProxy.system.listMethods()
```

This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

```python
ServerProxy.system.methodSignature(name)
```

This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply “string, array”. If it expects three integers and returns a string, its signature is “string, int, int, int”.

If no signature is defined for the method, a non-array value is
returned. In Python this means that the type of the returned value will be something other than list.

ServerProxy.system.methodHelp(name)
This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.
20.23.2. Boolean Objects

This class may be initialized from any Python value; the instance returned depends only on its truth value. It supports various Python operators through \_\_cmp\_\(), \_\_repr\_\(), \_\_int\_\(), and \_\_nonzero\_\() methods, all implemented in the obvious ways.

It also has the following method, supported mainly for internal use by the unmarshalling code:

Boolean. \texttt{encode(out)}

Write the XML-RPC encoding of this Boolean item to the output stream object.

A working example follows. The server code:

```python
import xmlrpclib
from SimpleXMLRPCServer import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(('localhost', 8000))
print "Listening on port 8000..."
server.register_function(is_even, "is_even")
server.serve_forever()
```

The client code for the preceding server:

```python
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
print "3 is even: %s" % str(proxy.is_even(3))
print "100 is even: %s" % str(proxy.is_even(100))
```
class xmlrpclib.DateTime

This class may be initialized with seconds since the epoch, a time tuple, an ISO 8601 time/date string, or a `datetime.datetime` instance. It has the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

**decode**(string)

Accept a string as the instance’s new time value.

**encode**(out)

Write the XML-RPC encoding of this `DateTime` item to the `out` stream object.

It also supports certain of Python’s built-in operators through `__cmp__()` and `__repr__()` methods.

A working example follows. The server code:

```python
import datetime
from SimpleXMLRPCServer import SimpleXMLRPCServer
import xmlrpclib

def today():
    today = datetime.datetime.today()
    return xmlrpclib.DateTime(today)

server = SimpleXMLRPCServer(('localhost', 8000))
print "Listening on port 8000..."
server.register_function(today, "today")
server.serve_forever()
```

The client code for the preceding server:

```python
import xmlrpclib
import datetime
```
proxy = xmlrpcclient.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print "Today: %s" % converted.strftime("%d.%m.%Y, %H:%M")
20.23.4. Binary Objects

class xmlrpclib.Binary

This class may be initialized from string data (which may include NULs). The primary access to the content of a Binary object is provided by an attribute:

data

The binary data encapsulated by the Binary instance. The data is provided as an 8-bit string.

Binary objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(string)

Accept a base64 string and decode it as the instance’s new data.

encode(out)

Write the XML-RPC base 64 encoding of this binary item to the out stream object.

The encoded data will have newlines every 76 characters as per RFC 2045 section 6.8, which was the de facto standard base64 specification when the XML-RPC spec was written.

It also supports certain of Python’s built-in operators through a __cmp__() method.

Example usage of the binary objects. We’re going to transfer an image over XMLRPC:

```python
from SimpleXMLRPCServer import SimpleXMLRPCServer
import xmlrpclib
```
def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpclib.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_function(python_logo, 'python_logo')

server.serve_forever()

The client gets the image and saves it to a file:

import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
20.23.5. Fault Objects

class xmlrpclib.Fault
    A Fault object encapsulates the content of an XML-RPC fault tag. Fault objects have the following attributes:

    faultCode
        A string indicating the fault type.

    faultString
        A string containing a diagnostic message associated with the fault.

In the following example we're going to intentionally cause a Fault by returning a complex type object. The server code:

```python
from SimpleXMLRPCServer import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(('localhost', 8000))
print "Listening on port 8000..."
server.register_function(add, 'add')
server.serve_forever()
```

The client code for the preceding server:

```python
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpclib.Fault as err:
    print "A fault occurred"
print "Fault code: %d" % err.faultCode
print "Fault string: %s" % err.faultString
20.23.6. ProtocolError Objects

```python
import xmlrpc
#
# create a ServerProxy with a URI that doesn't respond to XMLRPC
proxy = xmlrpc.ServerProxy("http://www.google.com/")

try:
    proxy.some_method()
except xmlrpc.ProtocolError as err:
    print "A protocol error occurred"
    print "URL: %s" % err.url
    print "HTTP/HTTPS headers: %s" % err.headers
    print "Error code: %d" % err.errcode
    print "Error message: %s" % err.errmsg
```
20.23.7. MultiCall Objects

New in version 2.4.

The `MultiCall` object provides a way to encapsulate multiple calls to a remote server into a single request [1].

```python
class xmlrpclib.MultiCall(server):
    Create an object used to boxcar method calls. server is the eventual target of the call. Calls can be made to the result object, but they will immediately return None, and only store the call name and parameters in the MultiCall object. Calling the object itself causes all stored calls to be transmitted as a single system.multicall request. The result of this call is a generator; iterating over this generator yields the individual results.
```

A usage example of this class follows. The server code

```python
from SimpleXMLRPCServer import SimpleXMLRPCServer

def add(x, y):
    return x+y

def subtract(x, y):
    return x-y

def multiply(x, y):
    return x*y

def divide(x, y):
    return x/y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(('localhost', 8000))
print "Listening on port 8000..."
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
```
server.register_function(divide, 'divide')
server.serve_forever()

The client code for the preceding server:

```python
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
multicall = xmlrpclib.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print "7+3=\%d, 7-3=\%d, 7*3=\%d, 7/3=\%d" % tuple(result)
```
20.23.8. Convenience Functions

**xmlrpclib.**boolean**(value)**

Convert any Python value to one of the XML-RPC Boolean constants, True or False.

**xmlrpclib.**dumps**(params[, methodname[, methodresponse[, encoding[, allow_none]]]])**

Convert *params* into an XML-RPC request. or into a response if *methodresponse* is true. *params* can be either a tuple of arguments or an instance of the **Fault** exception class. If *methodresponse* is true, only a single value can be returned, meaning that *params* must be of length 1. *encoding*, if supplied, is the encoding to use in the generated XML; the default is UTF-8. Python’s None value cannot be used in standard XML-RPC; to allow using it via an extension, provide a true value for *allow_none*.

**xmlrpclib.**loads**(data[, use_datetime]]**

Convert an XML-RPC request or response into Python objects, a (params, methodname). *params* is a tuple of argument; *methodname* is a string, or None if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a **Fault** exception. The use_datetime flag can be used to cause date/time values to be presented as datetime.datetime objects; this is false by default.

*Changed in version 2.5:* The use_datetime flag was added.
20.23.9. Example of Client Usage

```python
# simple test program (from the XML-RPC specification)
from xmlrpclib import ServerProxy, Error

# server = ServerProxy("http://localhost:8000")  # local server
server = ServerProxy("http://betty.userland.com")

print server

try:
    print server.examples.getStateName(41)
except Error as v:
    print "ERROR", v
```

To access an XML-RPC server through a HTTP proxy, you need to define a custom transport. The following example shows how:

```python
import xmlrpclib, httplib

class ProxiedTransport(xmlrpclib.Transport):
    def set_proxy(self, proxy):
        self.proxy = proxy

    def make_connection(self, host):
        self.realhost = host
        h = httplib.HTTPConnection(self.proxy)
        return h

    def send_request(self, connection, handler, request_body):
        connection.putrequest("POST", 'http://%s%s' % (self.realhost, request_body))

    def send_host(self, connection, host):
        connection.putheader('Host', self.realhost)

p = ProxiedTransport()
p.set_proxy('proxy-server:8080')
print server.currentTime.getCurrentTime()
```
20.23.10. Example of Client and Server Usage

See SimpleXMLRPCServer Example.

Footnotes

[1] This approach has been first presented in a discussion on xmlrpc.com.
20. Internet Protocols and Support
20.24. SimpleXMLRPCServer — Basic XML-RPC server

Note: The SimpleXMLRPCServer module has been merged into xmlrpc.server in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

New in version 2.2.

Source code: Lib/SimpleXMLRPCServer.py

The SimpleXMLRPCServer module provides a basic server framework for XML-RPC servers written in Python. Servers can either be free standing, using SimpleXMLRPCServer, or embedded in a CGI environment, using CGIXMRLPCRequestHandler.

class SimpleXMLRPCServer.

SimpleXMLRPCServer(addr[, requestHandler[, logRequests[, allow_none[, encoding[, bind_and_activate]]]]]

Create a new server instance. This class provides methods for registration of functions that can be called by the XML-RPC protocol. The requestHandler parameter should be a factory for request handler instances; it defaults to SimpleXMLRPCRequestHandler. The addr and requestHandler parameters are passed to the SocketServer.TCPServer constructor. If logRequests is true (the default), requests will be logged; setting this parameter to false will turn off logging. The allow_none and encoding parameters are passed on to xmlrpclib and control the XML-RPC responses that will be returned from the server. The bind_and_activate parameter controls whether server_bind() and server_activate() are
called immediately by the constructor; it defaults to true. Setting it to false allows code to manipulate the `allow_reuse_address` class variable before the address is bound.

*Changed in version 2.5:* The `allow_none` and `encoding` parameters were added.

*Changed in version 2.6:* The `bind_and_activate` parameter was added.

class SimpleXMLRPCServer.CGIXMLRPCRequestHandler([`allow_none`,
`encoding`])
Create a new instance to handle XML-RPC requests in a CGI environment. The `allow_none` and `encoding` parameters are passed on to `xmlrpclib` and control the XML-RPC responses that will be returned from the server.

*New in version 2.3.*

*Changed in version 2.5:* The `allow_none` and `encoding` parameters were added.

class SimpleXMLRPCServer.SimpleXMLRPCRequestHandler
Create a new request handler instance. This request handler supports `POST` requests and modifies logging so that the `logRequests` parameter to the `SimpleXMLRPCServer` constructor parameter is honored.
20.24.1. SimpleXMLRPCServer Objects

The `SimpleXMLRPCServer` class is based on `SocketServer.TCPServer` and provides a means of creating simple, stand alone XML-RPC servers.

`SimpleXMLRPCServer.register_function(function[, name])`

Register a function that can respond to XML-RPC requests. If `name` is given, it will be the method name associated with `function`, otherwise `function.__name__` will be used. `name` can be either a normal or Unicode string, and may contain characters not legal in Python identifiers, including the period character.

`SimpleXMLRPCServer.register_instance(instance[, allow_dotted_names])`

Register an object which is used to expose method names which have not been registered using `register_function()`. If `instance` contains a `_dispatch()` method, it is called with the requested method name and the parameters from the request. Its API is `def _dispatch(self, method, params)` (note that `params` does not represent a variable argument list). If it calls an underlying function to perform its task, that function is called as `func(*params)`, expanding the parameter list. The return value from `_dispatch()` is returned to the client as the result. If `instance` does not have a `_dispatch()` method, it is searched for an attribute matching the name of the requested method.

If the optional `allow_dotted_names` argument is true and the instance does not have a `_dispatch()` method, then if the requested method name contains periods, each component of the method name is searched for individually, with the effect that
a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

**Warning:** Enabling the *allow_dotted_names* option allows intruders to access your module's global variables and may allow intruders to execute arbitrary code on your machine. Only use this option on a secure, closed network.

*Changed in version 2.3.5:* 2.4.1 *allow_dotted_names* was added to plug a security hole; prior versions are insecure.

**SimpleXMLRPCServer.** register_introspection_functions()

Registers the XML-RPC introspection functions *system.listMethods*, *system.methodHelp* and *system.methodSignature*.

*New in version 2.3.*

**SimpleXMLRPCServer.** register_multicall_functions()

Registers the XML-RPC multicall function *system.multicall*.

**SimpleXMLRPCRequestHandler.** rpc_paths

An attribute value that must be a tuple listing valid path portions of the URL for receiving XML-RPC requests. Requests posted to other paths will result in a 404 “no such page” HTTP error. If this tuple is empty, all paths will be considered valid. The default value is (’/’, ’/RPC2’).

*New in version 2.5.*

**SimpleXMLRPCRequestHandler.** encode_threshold

If this attribute is not None, responses larger than this value will be encoded using the *gzip* transfer encoding, if permitted by the client. The default is 1400 which corresponds roughly to a single
TCP packet.

New in version 2.7.

## 20.24.1.1. SimpleXMLRPCServer Example

Server code:

```python
from SimpleXMLRPCServer import SimpleXMLRPCServer
from SimpleXMLRPCServer import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
server = SimpleXMLRPCServer(('localhost', 8000),
                             requestHandler=RequestHandler)
server.register_introspection_functions()

# Register pow() function; this will use the value of
# pow.__name__ as the name, which is just 'pow'.
server.register_function(pow)

# Register a function under a different name
def adder_function(x, y):
    return x + y
server.register_function(adder_function, 'add')

# Register an instance; all the methods of the instance are
# published as XML-RPC methods (in this case, just 'div').
class MyFuncs:
    def div(self, x, y):
        return x // y
server.register_instance(MyFuncs())

# Run the server's main loop
server.serve_forever()
```

The following client code will call the methods made available by the preceding server:
import xmlrpclib

s = xmlrpclib.ServerProxy('http://localhost:8000')
print s.pow(2, 3)  # Returns 2**3 = 8
print s.add(2, 3)  # Returns 5
print s.div(5, 2)  # Returns 5//2 = 2

# Print list of available methods
print s.system.listMethods()

The following SimpleXMLRPCServer example is included in the module Lib/SimpleXMLRPCServer.py:

server = SimpleXMLRPCServer("localhost", 8000)
server.register_function(pow)
server.register_function(lambda x, y: x+y, 'add')
server.register_multicall_functions()
server.serve_forever()

This demo server can be run from the command line as:

python -m SimpleXMLRPCServer

Example client code which talks to the above server is included with Lib/xmlrpclib.py:

server = ServerProxy("http://localhost:8000")
print server
multi = MultiCall(server)
multi.pow(2, 9)
multi.add(5, 1)
multi.add(24, 11)
try:
    for response in multi():
        print response
except Error, v:
    print "ERROR", v

And the client can be invoked directly using the following command:

python -m xmlrpclib
20.24.2. CGIXMLRPCRequestHandler

The **CGIXMLRPCRequestHandler** class can be used to handle XML-RPC requests sent to Python CGI scripts.

**CGIXMLRPCRequestHandler.register_function(function[, name])**

Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with function, otherwise *function.__name__* will be used. *name* can be either a normal or Unicode string, and may contain characters not legal in Python identifiers, including the period character.

**CGIXMLRPCRequestHandler.register_instance(instance)**

Register an object which is used to expose method names which have not been registered using *register_function()*. If instance contains a *_dispatch()* method, it is called with the requested method name and the parameters from the request; the return value is returned to the client as the result. If instance does not have a *_dispatch()* method, it is searched for an attribute matching the name of the requested method; if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

**CGIXMLRPCRequestHandler.register_introspection_functions()**

Register the XML-RPC introspection functions *system.listMethods*, *system.methodHelp* and *system.methodSignature*.

**CGIXMLRPCRequestHandler.register_multicall_functions()**
Register the XML-RPC multicall function `system.multicall`.

CGIXMLRPCRequestHandler. **handle_request**([*request_text = None*])

Handle an XML-RPC request. If `request_text` is given, it should be the POST data provided by the HTTP server, otherwise the contents of stdin will be used.

Example:

```python
class MyFuncs:
    def div(self, x, y): return x // y

handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```
20.25. DocXMLRPCServer — Self-documenting XML-RPC server

**Note:** The DocXMLRPCServer module has been merged into xmlrpc.server in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

*New in version 2.3.*

The DocXMLRPCServer module extends the classes found in SimpleXMLRPCServer to serve HTML documentation in response to HTTP GET requests. Servers can either be free standing, using DocXMLRPCServer, or embedded in a CGI environment, using DocCGIXMLRPCRequestHandler.

```python
class DocXMLRPCServer: DocXMLRPCServer(addr[, requestHandler[, logRequests[, allow_none[, encoding[, bind_and_activate]]]]]])

Create a new server instance. All parameters have the same meaning as for SimpleXMLRPCServer.SimpleXMLRPCServer;
requestHandler defaults to DocXMLRPCRequestHandler.

class DocXMLRPCServer: DocCGIXMLRPCRequestHandler

Create a new instance to handle XML-RPC requests in a CGI environment.

class DocXMLRPCServer: DocXMLRPCRequestHandler

Create a new request handler instance. This request handler supports XML-RPC POST requests, documentation GET requests, and modifies logging so that the logRequests parameter to the DocXMLRPCServer constructor parameter is honored.
The `DocXMLRPCServer` class is derived from `SimpleXMLRPCServer.SimpleXMLRPCServer` and provides a means of creating self-documenting, stand alone XML-RPC servers. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

`DocXMLRPCServer.set_server_title(server_title)`

Set the title used in the generated HTML documentation. This title will be used inside the HTML “title” element.

`DocXMLRPCServer.set_server_name(server_name)`

Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a “h1” element.

`DocXMLRPCServer.set_server_documentation(server_documentation)`

Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.
20.25.2. DocCGIXMLRPCRequestHandler

The `DocCGIXMLRPCRequestHandler` class is derived from `SimpleXMLRPCServer.CGIXMLRPCRequestHandler` and provides a means of creating self-documenting, XML-RPC CGI scripts. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

`DocCGIXMLRPCRequestHandler.set_server_title(server_title)`

Set the title used in the generated HTML documentation. This title will be used inside the HTML “title” element.

`DocCGIXMLRPCRequestHandler.set_server_name(server_name)`

Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a “h1” element.

`DocCGIXMLRPCRequestHandler.set_server_documentation(server_documentation)`

Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.
21. Multimedia Services

The modules described in this chapter implement various algorithms or interfaces that are mainly useful for multimedia applications. They are available at the discretion of the installation. Here’s an overview:

- **21.1. audioop** — Manipulate raw audio data
- **21.2. imageop** — Manipulate raw image data
- **21.3. aifc** — Read and write AIFF and AIFC files
- **21.4. sunau** — Read and write Sun AU files
  - **21.4.1. AU_read Objects**
  - **21.4.2. AU_write Objects**
- **21.5. wave** — Read and write WAV files
  - **21.5.1. Wave_read Objects**
  - **21.5.2. Wave_write Objects**
- **21.6. chunk** — Read IFF chunked data
- **21.7. colorsys** — Conversions between color systems
- **21.8. imghdr** — Determine the type of an image
- **21.9. sndhdr** — Determine type of sound file
- **21.10. ossaudiodev** — Access to OSS-compatible audio devices
  - **21.10.1. Audio Device Objects**
  - **21.10.2. Mixer Device Objects**
21.1. audioop — Manipulate raw audio data

The audioop module contains some useful operations on sound fragments. It operates on sound fragments consisting of signed integer samples 8, 16 or 32 bits wide, stored in Python strings. This is the same format as used by the al and sunaudiodev modules. All scalar items are integers, unless specified otherwise.

This module provides support for a-LAW, u-LAW and Intel/DVI ADPCM encodings.

A few of the more complicated operations only take 16-bit samples, otherwise the sample size (in bytes) is always a parameter of the operation.

The module defines the following variables and functions:

_exception audioop.**error**

This exception is raised on all errors, such as unknown number of bytes per sample, etc.

audioop.**add**(fragment1, fragment2, width)

Return a fragment which is the addition of the two samples passed as parameters. width is the sample width in bytes, either 1, 2 or 4. Both fragments should have the same length. Samples are truncated in case of overflow.

audioop.**adpcm2lin**(adpcmfragment, width, state)

Decode an Intel/DVI ADPCM coded fragment to a linear fragment. See the description of lin2adpcm() for details on ADPCM coding. Return a tuple (sample, newstate) where the sample has the width specified in width.
audioop. **alaw2lin**(fragment, width)

Convert sound fragments in a-LAW encoding to linearly encoded sound fragments. a-LAW encoding always uses 8 bits samples, so width refers only to the sample width of the output fragment here.

*New in version 2.5.*

audioop. **avg**(fragment, width)

Return the average over all samples in the fragment.

audioop. **avgpp**(fragment, width)

Return the average peak-peak value over all samples in the fragment. No filtering is done, so the usefulness of this routine is questionable.

audioop. **bias**(fragment, width, bias)

Return a fragment that is the original fragment with a bias added to each sample. Samples wrap around in case of overflow.

audioop. **cross**(fragment, width)

Return the number of zero crossings in the fragment passed as an argument.

audioop. **findfactor**(fragment, reference)

Return a factor $F$ such that \( \text{rms}(\text{add}(\text{fragment}, \text{mul}(\text{reference}, -F))) \) is minimal, i.e., return the factor with which you should multiply reference to make it match as well as possible to fragment. The fragments should both contain 2-byte samples.

The time taken by this routine is proportional to \( \text{len(fragment)} \).

audioop. **findfit**(fragment, reference)

Try to match reference as well as possible to a portion of fragment (which should be the longer fragment). This is
(conceptually) done by taking slices out of fragment, using `findfactor()` to compute the best match, and minimizing the result. The fragments should both contain 2-byte samples. Return a tuple `(offset, factor)` where offset is the (integer) offset into fragment where the optimal match started and factor is the (floating-point) factor as per `findfactor()`.

`audioop.findmax(fragment, length)`

Search fragment for a slice of length length samples (not bytes!) with maximum energy, i.e., return i for which \[ \text{rms}(\text{fragment}[i\times2: (i+\text{length})\times2]) \] is maximal. The fragments should both contain 2-byte samples.

The routine takes time proportional to \( \text{len(fragment)} \).

`audioop.getsample(fragment, width, index)`

Return the value of sample index from the fragment.

`audioop.lin2adpcm(fragment, width, state)`

Convert samples to 4 bit Intel/DVI ADPCM encoding. ADPCM coding is an adaptive coding scheme, whereby each 4 bit number is the difference between one sample and the next, divided by a (varying) step. The Intel/DVI ADPCM algorithm has been selected for use by the IMA, so it may well become a standard.

state is a tuple containing the state of the coder. The coder returns a tuple `(adpcmfrag, newstate)`, and the newstate should be passed to the next call of `lin2adpcm()`. In the initial call, `None` can be passed as the state. adpcmfrag is the ADPCM coded fragment packed 2 4-bit values per byte.

`audioop.lin2alaw(fragment, width)`

Convert samples in the audio fragment to a-LAW encoding and return this as a Python string. a-LAW is an audio encoding format
whereby you get a dynamic range of about 13 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

*New in version 2.5.*

**audioop.** `lin2lin(fragment, width, newwidth)`

Convert samples between 1-, 2- and 4-byte formats.

**Note:** In some audio formats, such as .WAV files, 16 and 32 bit samples are signed, but 8 bit samples are unsigned. So when converting to 8 bit wide samples for these formats, you need to also add 128 to the result:

```python
new_frames = audioop.lin2lin(frames, old_width, 1)
new_frames = audioop.bias(new_frames, 1, 128)
```

The same, in reverse, has to be applied when converting from 8 to 16 or 32 bit width samples.

**audioop.** `lin2ulaw(fragment, width)`

Convert samples in the audio fragment to u-LAW encoding and return this as a Python string. u-LAW is an audio encoding format whereby you get a dynamic range of about 14 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

**audioop.** `max(fragment, width)`

Return the maximum of the *absolute value* of all samples in a fragment.

**audioop.** `maxpp(fragment, width)`

Return the maximum peak-peak value in the sound fragment.

**audioop.** `minmax(fragment, width)`

Return a tuple consisting of the minimum and maximum values of all samples in the sound fragment.
audioop.mul(fragment, width, factor)

Return a fragment that has all samples in the original fragment multiplied by the floating-point value `factor`. Samples are truncated in case of overflow.

audioop.ratecv(fragment, width, nchannels, inrate, outrate, state[, weightA[, weightB]])

Convert the frame rate of the input fragment.

`state` is a tuple containing the state of the converter. The converter returns a tuple (newfragment, newstate), and `newstate` should be passed to the next call of `ratecv()`. The initial call should pass `None` as the state.

The `weightA` and `weightB` arguments are parameters for a simple digital filter and default to 1 and 0 respectively.

audioop.reverse(fragment, width)

Reverse the samples in a fragment and returns the modified fragment.

audioop.rms(fragment, width)

Return the root-mean-square of the fragment, i.e.\n\[
\text{sqrt}\left(\sum S_i^2/n\right).
\]

This is a measure of the power in an audio signal.

audioop.tomono(fragment, width, lfactor, rfactor)

Convert a stereo fragment to a mono fragment. The left channel is multiplied by `lfactor` and the right channel by `rfactor` before adding the two channels to give a mono signal.

audioop.tostereo(fragment, width, lfactor, rfactor)

Generate a stereo fragment from a mono fragment. Each pair of samples in the stereo fragment are computed from the mono
sample, whereby left channel samples are multiplied by *lfactor* and right channel samples by *rfactor*.

`audioop.ulaw2lin(fragment, width)`

Convert sound fragments in u-LAW encoding to linearly encoded sound fragments. u-LAW encoding always uses 8 bits samples, so *width* refers only to the sample width of the output fragment here.

Note that operations such as `mul()` or `max()` make no distinction between mono and stereo fragments, i.e. all samples are treated equal. If this is a problem the stereo fragment should be split into two mono fragments first and recombined later. Here is an example of how to do that:

```python
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
    lsample = audioop.mul(lsample, width, lfactor)
    rsample = audioop.mul(rsample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

If you use the ADPCM coder to build network packets and you want your protocol to be stateless (i.e. to be able to tolerate packet loss) you should not only transmit the data but also the state. Note that you should send the *initial* state (the one you passed to `lin2adpcm()`) along to the decoder, not the final state (as returned by the coder). If you want to use `struct.Struct` to store the state in binary you can code the first element (the predicted value) in 16 bits and the second (the delta index) in 8.

The ADPCM coders have never been tried against other ADPCM coders, only against themselves. It could well be that I misinterpreted the standards in which case they will not be interoperable with the respective standards.
The `find*()` routines might look a bit funny at first sight. They are primarily meant to do echo cancellation. A reasonably fast way to do this is to pick the most energetic piece of the output sample, locate that in the input sample and subtract the whole output sample from the input sample:

```python
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)  # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)],
    # out_test)
    prefill = '\0'*((pos+ipos)*2)
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefill + audioop.mul(outputdata, 2, -factor)
    return audioop.add(inputdata, outputdata, 2)
```
21.2. imageop — Manipulate raw image data

Deprecation since version 2.6: The imageop module has been removed in Python 3.

The imageop module contains some useful operations on images. It operates on images consisting of 8 or 32 bit pixels stored in Python strings. This is the same format as used by gl.lrectwrite() and the imgfile module.

The module defines the following variables and functions:

**exception imageop.error**

This exception is raised on all errors, such as unknown number of bits per pixel, etc.

**imageop.crop(image, psize, width, height, x0, y0, x1, y1)**

Return the selected part of image, which should be width by height in size and consist of pixels of psize bytes. x0, y0, x1 and y1 are like the gl.lrectread() parameters, i.e. the boundary is included in the new image. The new boundaries need not be inside the picture. Pixels that fall outside the old image will have their value set to zero. If x0 is bigger than x1 the new image is mirrored. The same holds for the y coordinates.

**imageop.scale(image, psize, width, height, newwidth, newheight)**

Return image scaled to size newwidth by newheight. No interpolation is done, scaling is done by simple-minded pixel duplication or removal. Therefore, computer-generated images or dithered images will not look nice after scaling.
imageop.tovideo(image, psize, width, height)
Run a vertical low-pass filter over an image. It does so by computing each destination pixel as the average of two vertically-aligned source pixels. The main use of this routine is to forestall excessive flicker if the image is displayed on a video device that uses interlacing, hence the name.

imageop.grey2mono(image, width, height, threshold)
Convert an 8-bit deep greyscale image to a 1-bit deep image by thresholding all the pixels. The resulting image is tightly packed and is probably only useful as an argument to mono2grey().

imageop.dither2mono(image, width, height)
Convert an 8-bit greyscale image to a 1-bit monochrome image using a (simple-minded) dithering algorithm.

imageop.mono2grey(image, width, height, p0, p1)
Convert a 1-bit monochrome image to an 8 bit greyscale or color image. All pixels that are zero-valued on input get value \( p0 \) on output and all one-value input pixels get value \( p1 \) on output. To convert a monochrome black-and-white image to greyscale pass the values 0 and 255 respectively.

imageop.grey2grey4(image, width, height)
Convert an 8-bit greyscale image to a 4-bit greyscale image without dithering.

imageop.grey2grey2(image, width, height)
Convert an 8-bit greyscale image to a 2-bit greyscale image without dithering.

imageop.dither2grey2(image, width, height)
Convert an 8-bit greyscale image to a 2-bit greyscale image with dithering. As for dither2mono(), the dithering algorithm is currently very simple.
imageop. **grey42grey**(image, width, height)
Convert a 4-bit greyscale image to an 8-bit greyscale image.

imageop. **grey22grey**(image, width, height)
Convert a 2-bit greyscale image to an 8-bit greyscale image.

imageop. **backward_compatible**
If set to 0, the functions in this module use a non-backward compatible way of representing multi-byte pixels on little-endian systems. The SGI for which this module was originally written is a big-endian system, so setting this variable will have no effect. However, the code wasn’t originally intended to run on anything else, so it made assumptions about byte order which are not universal. Setting this variable to 0 will cause the byte order to be reversed on little-endian systems, so that it then is the same as on big-endian systems.
21. Multimedia Services
21.3. **aifc** — Read and write AIFF and AIFC files

**Source code:** Lib/aifc.py

This module provides support for reading and writing AIFF and AIF-C files. AIFF is Audio Interchange File Format, a format for storing digital audio samples in a file. AIFF-C is a newer version of the format that includes the ability to compress the audio data.

**Note:** Some operations may only work under IRIX; these will raise `ImportError` when attempting to import the `cl` module, which is only available on IRIX.

Audio files have a number of parameters that describe the audio data. The sampling rate or frame rate is the number of times per second the sound is sampled. The number of channels indicate if the audio is mono, stereo, or quadro. Each frame consists of one sample per channel. The sample size is the size in bytes of each sample. Thus a frame consists of `nchannels*samplesize` bytes, and a second's worth of audio consists of `nchannels*samplesize*framerate` bytes.

For example, CD quality audio has a sample size of two bytes (16 bits), uses two channels (stereo) and has a frame rate of 44,100 frames/second. This gives a frame size of 4 bytes (2*2), and a second's worth occupies 2*2*44100 bytes (176,400 bytes).

Module **aifc** defines the following function:

```python
aifc.open(file[, mode])
```

Open an AIFF or AIFF-C file and return an object instance with
methods that are described below. The argument file is either a string naming a file or a file object. mode must be 'r' or 'rb' when the file must be opened for reading, or 'w' or 'wb' when the file must be opened for writing. If omitted, file.mode is used if it exists, otherwise 'rb' is used. When used for writing, the file object should be seekable, unless you know ahead of time how many samples you are going to write in total and use writeframesraw() and setnframes().

Objects returned by open() when a file is opened for reading have the following methods:

aifc.getnchannels()
   Return the number of audio channels (1 for mono, 2 for stereo).

aifc.getsampwidth()
   Return the size in bytes of individual samples.

aifc.getframerate()
   Return the sampling rate (number of audio frames per second).

aifc.getnframes()
   Return the number of audio frames in the file.

aifc.getcomptype()
   Return a four-character string describing the type of compression used in the audio file. For AIFF files, the returned value is 'NONE'.

aifc.getcompname()
   Return a human-readable description of the type of compression used in the audio file. For AIFF files, the returned value is 'not compressed'.

aifc.getparams()
   Return a tuple consisting of all of the above values in the above
order.

aifc.getmarkers()
Return a list of markers in the audio file. A marker consists of a tuple of three elements. The first is the mark ID (an integer), the second is the mark position in frames from the beginning of the data (an integer), the third is the name of the mark (a string).

aifc.getmark(id)
Return the tuple as described in getmarkers() for the mark with the given id.

aifc.readframes(nframes)
Read and return the next nframes frames from the audio file. The returned data is a string containing for each frame the uncompressed samples of all channels.

aifc.rewind()
Rewind the read pointer. The next readframes() will start from the beginning.

aifc.setpos(pos)
Seek to the specified frame number.

aifc.tell()
Return the current frame number.

aifc.close()
Close the AIFF file. After calling this method, the object can no longer be used.

Objects returned by open() when a file is opened for writing have all the above methods, except for readframes() and setpos(). In addition the following methods exist. The get*() methods can only be called after the corresponding set*() methods have been called.
Before the first `writeframes()` or `writeframesraw()`, all parameters except for the number of frames must be filled in.

**aifc.aiff()**
Create an AIFF file. The default is that an AIFF-C file is created, unless the name of the file ends in `.aiff` in which case the default is an AIFF file.

**aifc.aifc()**
Create an AIFF-C file. The default is that an AIFF-C file is created, unless the name of the file ends in `.aiff` in which case the default is an AIFF file.

**aifc.setnchannels(nchannels)**
Specify the number of channels in the audio file.

**aifc.setsampwidth(width)**
Specify the size in bytes of audio samples.

**aifc.setframerate(rate)**
Specify the sampling frequency in frames per second.

**aifc.setnframes(nframes)**
Specify the number of frames that are to be written to the audio file. If this parameter is not set, or not set correctly, the file needs to support seeking.

**aifc.setcomptype(type, name)**
Specify the compression type. If not specified, the audio data will not be compressed. In AIFF files, compression is not possible. The name parameter should be a human-readable description of the compression type, the type parameter should be a four-character string. Currently the following compression types are supported: NONE, ULAW, ALAW, G722.
aifc.setparams(nchannels, sampwidth, framerate, comptype, compname)

Set all the above parameters at once. The argument is a tuple consisting of the various parameters. This means that it is possible to use the result of a getparams() call as argument to setparams().

aifc.setmark(id, pos, name)

Add a mark with the given id (larger than 0), and the given name at the given position. This method can be called at any time before close().

aifc.tell()

Return the current write position in the output file. Useful in combination with setmark().

aifc.writeframes(data)

Write data to the output file. This method can only be called after the audio file parameters have been set.

aifc.writeframesraw(data)

Like writeframes(), except that the header of the audio file is not updated.

aifc.close()

Close the AIFF file. The header of the file is updated to reflect the actual size of the audio data. After calling this method, the object can no longer be used.
21.4. **sunau** — Read and write Sun AU files

Source code: Lib/sunau.py

The **sunau** module provides a convenient interface to the Sun AU sound format. Note that this module is interface-compatible with the modules **aifc** and **wave**.

An audio file consists of a header followed by the data. The fields of the header are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>magic word</td>
<td>The four bytes <code>.snd</code>.</td>
</tr>
<tr>
<td>header size</td>
<td>Size of the header, including info, in bytes.</td>
</tr>
<tr>
<td>data size</td>
<td>Physical size of the data, in bytes.</td>
</tr>
<tr>
<td>encoding</td>
<td>Indicates how the audio samples are encoded.</td>
</tr>
<tr>
<td>sample rate</td>
<td>The sampling rate.</td>
</tr>
<tr>
<td># of channels</td>
<td>The number of channels in the samples.</td>
</tr>
<tr>
<td>info</td>
<td>ASCII string giving a description of the audio file (padded with null bytes).</td>
</tr>
</tbody>
</table>

Apart from the info field, all header fields are 4 bytes in size. They are all 32-bit unsigned integers encoded in big-endian byte order.

The **sunau** module defines the following functions:

```python
sunau.open(file, mode)
```

If `file` is a string, open the file by that name, otherwise treat it as a seekable file-like object. `mode` can be any of

```
'r'
```
Read only mode.

'w'

Write only mode.

Note that it does not allow read/write files.

A mode of 'r' returns an AU_read object, while a mode of 'w' or 'wb' returns an AU_write object.

sunau.openfp(file, mode)

A synonym for open(), maintained for backwards compatibility.

The sunau module defines the following exception:

exception sunau.Error

An error raised when something is impossible because of Sun AU specs or implementation deficiency.

The sunau module defines the following data items:

sunau.AUDIO_FILE_MAGIC

An integer every valid Sun AU file begins with, stored in big-endian form. This is the string .snd interpreted as an integer.

sunau.AUDIO_FILE_ENCODING_MULAW_8
sunau.AUDIO_FILE_ENCODING_LINEAR_8
sunau.AUDIO_FILE_ENCODING_LINEAR_16
sunau.AUDIO_FILE_ENCODING_LINEAR_24
sunau.AUDIO_FILE_ENCODING_LINEAR_32
sunau.AUDIO_FILE_ENCODING_ALAW_8

Values of the encoding field from the AU header which are supported by this module.

sunau.AUDIO_FILE_ENCODING_FLOAT
sunau.AUDIO_FILE_ENCODING_DOUBLE
sunau.AUDIO_FILE_ENCODING_ADPCM_G721
sunau.AUDIO_FILE_ENCODING_ADPCM_G722
sunau. \texttt{AUDIO_FILE_ENCODING_ADPCM_G723_3}

sunau. \texttt{AUDIO_FILE_ENCODING_ADPCM_G723_5}

Additional known values of the encoding field from the AU header, but which are not supported by this module.
21.4.1. AU_read Objects

AU_read objects, as returned by `open()` above, have the following methods:

**AU_read.close()**
Close the stream, and make the instance unusable. (This is called automatically on deletion.)

**AU_read.getnchannels()**
Returns number of audio channels (1 for mono, 2 for stereo).

**AU_read.getsampwidth()**
Returns sample width in bytes.

**AU_read.getframerate()**
Returns sampling frequency.

**AU_read.getnframes()**
Returns number of audio frames.

**AU_read.getcomptype()**
Returns compression type. Supported compression types are 'ULAW', 'ALAW' and 'NONE'.

**AU_read.getcompname()**
Human-readable version of `getcomptype()`. The supported types have the respective names 'CCITT G.711 u-law', 'CCITT G.711 A-law' and 'not compressed'.

**AU_read.getparams()**
Returns a tuple `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, equivalent to output of the `get*()` methods.
**AU_read.readframes(n)**
Reads and returns at most n frames of audio, as a string of bytes. The data will be returned in linear format. If the original data is in u-LAW format, it will be converted.

**AU_read.rewind()**
Rewind the file pointer to the beginning of the audio stream.

The following two methods define a term “position” which is compatible between them, and is otherwise implementation dependent.

**AU_read.setpos(pos)**
Set the file pointer to the specified position. Only values returned from `tell()` should be used for `pos`.

**AU_read.tell()**
Return current file pointer position. Note that the returned value has nothing to do with the actual position in the file.

The following two functions are defined for compatibility with the aifc, and don’t do anything interesting.

**AU_read.getmarkers()**
Returns None.

**AU_read.getmark(id)**
Raise an error.
21.4.2. AU_write Objects

AU_write objects, as returned by `open()` above, have the following methods:

**AU_write.setnchannels(n)**
Set the number of channels.

**AU_write.setsampwidth(n)**
Set the sample width (in bytes.)

**AU_write.setframerate(n)**
Set the frame rate.

**AU_write.setnframes(n)**
Set the number of frames. This can be later changed, when and if more frames are written.

**AU_write.setcomptype(type, name)**
Set the compression type and description. Only 'NONE' and 'ULAW' are supported on output.

**AU_write.setparams(tuple)**
The tuple should be `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, with values valid for the `set*()` methods. Set all parameters.

**AU_write.tell()**
Return current position in the file, with the same disclaimer for the `AU_read.tell()` and `AU_read.setpos()` methods.

**AU_write.writeframesraw(data)**
Write audio frames, without correcting `nframes`. 
AU_write.writeframes(data)
Write audio frames and make sure nframes is correct.

AU_write.close()
Make sure nframes is correct, and close the file.

This method is called upon deletion.

Note that it is invalid to set any parameters after calling writeframes() or writeframesraw().
The `wave` module provides a convenient interface to the WAV sound format. It does not support compression/decompression, but it does support mono/stereo.

The `wave` module defines the following function and exception:

```
wave.open(file[, mode])
```

If `file` is a string, open the file by that name, otherwise treat it as a seekable file-like object. `mode` can be any of

- `'r'` or `'rb'`
  
  Read only mode.

- `'w'` or `'wb'`
  
  Write only mode.

Note that it does not allow read/write WAV files.

A `mode` of `'r'` or `'rb'` returns a `Wave_read` object, while a `mode` of `'w'` or `'wb'` returns a `Wave_write` object. If `mode` is omitted and a file-like object is passed as `file`, `file.mode` is used as the default value for `mode` (the `'b'` flag is still added if necessary).

If you pass in a file-like object, the `wave` object will not close it when its `close()` method is called; it is the caller’s responsibility to close the file object.

```
wave.openfp(file, mode)
```
A synonym for `open()`, maintained for backwards compatibility.

**Error**

An error raised when something is impossible because it violates the WAV specification or hits an implementation deficiency.
### 21.5.1. Wave_read Objects

Wave_read objects, as returned by `open()`, have the following methods:

Wave_read.

#### close()

Close the stream if it was opened by `wave`, and make the instance unusable. This is called automatically on object collection.

Wave_read.

#### getnchannels()

Returns number of audio channels (1 for mono, 2 for stereo).

Wave_read.

#### getsampwidth()

Returns sample width in bytes.

Wave_read.

#### getframerate()

Returns sampling frequency.

Wave_read.

#### getnframes()

Returns number of audio frames.

Wave_read.

#### getcomptype()

Returns compression type (`'NONE'` is the only supported type).

Wave_read.

#### getcompname()

Human-readable version of `getcomptype()`. Usually 'not compressed' parallels 'NONE'.

Wave_read.

#### getparams()

Returns a tuple `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, equivalent to output of the `get*()` methods.

Wave_read.

#### readframes(n)
Reads and returns at most \( n \) frames of audio, as a string of bytes.

`Wave_read.rewind()`
Rewind the file pointer to the beginning of the audio stream.

The following two methods are defined for compatibility with the `aiff` module, and don't do anything interesting.

`Wave_read.getmarkers()`
Returns `None`.

`Wave_read.getmark(id)`
Raise an error.

The following two methods define a term “position” which is compatible between them, and is otherwise implementation dependent.

`Wave_read.setpos(pos)`
Set the file pointer to the specified position.

`Wave_read.tell()`
Return current file pointer position.
21.5.2. Wave_write Objects

Wave_write objects, as returned by `open()`, have the following methods:

Wave_write.close()

Make sure `nframes` is correct, and close the file if it was opened by `wave`. This method is called upon object collection.

Wave_write.setnchannels(n)

Set the number of channels.

Wave_write.setsampwidth(n)

Set the sample width to `n` bytes.

Wave_write.setframerate(n)

Set the frame rate to `n`.

Wave_write.setnframes(n)

Set the number of frames to `n`. This will be changed later if more frames are written.

Wave_write.setcomptype(type, name)

Set the compression type and description. At the moment, only compression type `NONE` is supported, meaning no compression.

Wave_write.setparams(tuple)

The `tuple` should be `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, with values valid for the `set*()` methods. Sets all parameters.

Wave_write.tell()

Return current position in the file, with the same disclaimer for the `Wave_read.tell()` and `Wave_read.setpos()` methods.
Wave_write.\texttt{writeframesraw(\textit{data})}
Write audio frames, without correcting \textit{nframes}.

Wave_write.\texttt{writeframes(\textit{data})}
Write audio frames and make sure \textit{nframes} is correct.

Note that it is invalid to set any parameters after calling \texttt{writeframes()} or \texttt{writeframesraw()}, and any attempt to do so will raise \texttt{wave.Error}. 
21.6. chunk — Read IFF chunked data

This module provides an interface for reading files that use EA IFF 85 chunks. [1] This format is used in at least the Audio Interchange File Format (AIFF/AIFF-C) and the Real Media File Format (RMFF). The WAVE audio file format is closely related and can also be read using this module.

A chunk has the following structure:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Chunk ID</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Size of chunk in big-endian byte order, not</td>
</tr>
<tr>
<td></td>
<td></td>
<td>including the header</td>
</tr>
<tr>
<td>8</td>
<td>n</td>
<td>Data bytes, where n is the size given in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>preceding field</td>
</tr>
<tr>
<td>8 + n</td>
<td>0 or 1</td>
<td>Pad byte needed if n is odd and chunk</td>
</tr>
<tr>
<td></td>
<td></td>
<td>alignment is used</td>
</tr>
</tbody>
</table>

The ID is a 4-byte string which identifies the type of chunk.

The size field (a 32-bit value, encoded using big-endian byte order) gives the size of the chunk data, not including the 8-byte header.

Usually an IFF-type file consists of one or more chunks. The proposed usage of the `chunk` class defined here is to instantiate an instance at the start of each chunk and read from the instance until it reaches the end, after which a new instance can be instantiated. At the end of the file, creating a new instance will fail with an `EOFError` exception.

```
class chunk. Chunk(file[, align, bigendian, inclheader])
```
Class which represents a chunk. The file argument is expected to be a file-like object. An instance of this class is specifically allowed. The only method that is needed is read(). If the methods seek() and tell() are present and don’t raise an exception, they are also used. If these methods are present and raise an exception, they are expected to not have altered the object. If the optional argument align is true, chunks are assumed to be aligned on 2-byte boundaries. If align is false, no alignment is assumed. The default value is true. If the optional argument bigendian is false, the chunk size is assumed to be in little-endian order. This is needed for WAVE audio files. The default value is true. If the optional argument inclheader is true, the size given in the chunk header includes the size of the header. The default value is false.

A Chunk object supports the following methods:

getname()

Returns the name (ID) of the chunk. This is the first 4 bytes of the chunk.

getsize()

Returns the size of the chunk.

close()

Close and skip to the end of the chunk. This does not close the underlying file.

The remaining methods will raise IOError if called after the close() method has been called.

isatty()

Returns False.
**seek**(pos[, whence])

Set the chunk’s current position. The whence argument is optional and defaults to 0 (absolute file positioning); other values are 1 (seek relative to the current position) and 2 (seek relative to the file’s end). There is no return value. If the underlying file does not allow seek, only forward seeks are allowed.

**tell()**

Return the current position into the chunk.

**read([size])**

Read at most size bytes from the chunk (less if the read hits the end of the chunk before obtaining size bytes). If the size argument is negative or omitted, read all data until the end of the chunk. The bytes are returned as a string object. An empty string is returned when the end of the chunk is encountered immediately.

**skip()**

Skip to the end of the chunk. All further calls to **read()** for the chunk will return ‘’. If you are not interested in the contents of the chunk, this method should be called so that the file points to the start of the next chunk.

---

**Footnotes**

The `colorsys` module defines bidirectional conversions of color values between colors expressed in the RGB (Red Green Blue) color space used in computer monitors and three other coordinate systems: YIQ, HLS (Hue Lightness Saturation) and HSV (Hue Saturation Value). Coordinates in all of these color spaces are floating point values. In the YIQ space, the Y coordinate is between 0 and 1, but the I and Q coordinates can be positive or negative. In all other spaces, the coordinates are all between 0 and 1.


The `colorsys` module defines the following functions:

- `colorsys.rgb_to_yiq(r, g, b)`: Convert the color from RGB coordinates to YIQ coordinates.
- `colorsys.yiq_to_rgb(y, i, q)`: Convert the color from YIQ coordinates to RGB coordinates.
- `colorsys.rgb_to_hls(r, g, b)`: Convert the color from RGB coordinates to HLS coordinates.
- `colorsys.hls_to_rgb(h, l, s)`: Convert the color from HLS coordinates to RGB coordinates.
colorsys.\texttt{rgb\_to\_hsv}(r, g, b)

Convert the color from RGB coordinates to HSV coordinates.

colorsys.\texttt{hsv\_to\_rgb}(h, s, v)

Convert the color from HSV coordinates to RGB coordinates.

Example:

```python
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```
21.8. `imghdr` — Determine the type of an image

**Source code:** Lib/imghdr.py

The `imghdr` module determines the type of image contained in a file or byte stream.

The `imghdr` module defines the following function:

```python
imghdr.what(filename[, h])
```

Tests the image data contained in the file named by `filename`, and returns a string describing the image type. If optional `h` is provided, the `filename` is ignored and `h` is assumed to contain the byte stream to test.

The following image types are recognized, as listed below with the return value from `what()`:

<table>
<thead>
<tr>
<th>Value</th>
<th>Image format</th>
</tr>
</thead>
<tbody>
<tr>
<td>'rgb'</td>
<td>SGI ImgLib Files</td>
</tr>
<tr>
<td>'gif'</td>
<td>GIF 87a and 89a Files</td>
</tr>
<tr>
<td>'pbm'</td>
<td>Portable Bitmap Files</td>
</tr>
<tr>
<td>'pgm'</td>
<td>Portable Graymap Files</td>
</tr>
<tr>
<td>'ppm'</td>
<td>Portable Pixmap Files</td>
</tr>
<tr>
<td>'tiff'</td>
<td>TIFF Files</td>
</tr>
<tr>
<td>'rast'</td>
<td>Sun Raster Files</td>
</tr>
<tr>
<td>'xbm'</td>
<td>X Bitmap Files</td>
</tr>
<tr>
<td>'jpeg'</td>
<td>JPEG data in JFIF or Exif formats</td>
</tr>
<tr>
<td>'bmp'</td>
<td>BMP files</td>
</tr>
</tbody>
</table>
{'png': Portable Network Graphics}

New in version 2.5: Exif detection.

You can extend the list of file types `imghdr` can recognize by appending to this variable:

```
imghdr.tests
```

A list of functions performing the individual tests. Each function takes two arguments: the byte-stream and an open file-like object. When `what()` is called with a byte-stream, the file-like object will be `None`.

The test function should return a string describing the image type if the test succeeded, or `None` if it failed.

Example:

```python
>>> import imghdr
>>> imghdr.what('bass.gif')
'gif'
```
21.9. sndhdr — Determine type of sound file

Source code: Lib/sndhdr.py

The `sndhdr` provides utility functions which attempt to determine the type of sound data which is in a file. When these functions are able to determine what type of sound data is stored in a file, they return a tuple `(type, sampling_rate, channels, frames, bits_per_sample)`. The value for `type` indicates the data type and will be one of the strings 'aifc', 'aiff', 'au', 'hcom', 'sndr', 'sndt', 'voc', 'wav', '8svx', 'sb', 'ub', or 'ul'. The `sampling_rate` will be either the actual value or 0 if unknown or difficult to decode. Similarly, `channels` will be either the number of channels or 0 if it cannot be determined or if the value is difficult to decode. The value for `frames` will be either the number of frames or -1. The last item in the tuple, `bits_per_sample`, will either be the sample size in bits or 'A' for A-LAW or 'U' for u-LAW.

```
sndhdr.what(filename)
```

Determines the type of sound data stored in the file `filename` using `whathdr()`. If it succeeds, returns a tuple as described above, otherwise `None` is returned.

```
sndhdr.whathdr(filename)
```

Determines the type of sound data stored in a file based on the file header. The name of the file is given by `filename`. This function returns a tuple as described above on success, or `None`.
21.10. **ossaudiodev — Access to OSS-compatible audio devices**

*New in version 2.3.*

This module allows you to access the OSS (Open Sound System) audio interface. OSS is available for a wide range of open-source and commercial Unices, and is the standard audio interface for Linux and recent versions of FreeBSD.

**See also:**

*Open Sound System Programmer’s Guide*

the official documentation for the OSS C API

The module defines a large number of constants supplied by the OSS device driver; see `<sys/soundcard.h>` on either Linux or FreeBSD for a listing.

**ossaudiodev** defines the following variables and functions:

**exception** `ossaudiodev.OSSAudioError`

This exception is raised on certain errors. The argument is a string describing what went wrong.

(If `ossaudiodev` receives an error from a system call such as `open()`, `write()`, or `ioctl()`, it raises `IOError`. Errors detected directly by `ossaudiodev` result in `OSSAudioError`.)

(For backwards compatibility, the exception class is also available as `ossaudiodev.error`.)

`ossaudiodev.open(mode)`
ossaudiodev. `open(device, mode)`

Open an audio device and return an OSS audio device object. This object supports many file-like methods, such as `read()`, `write()`, and `fileno()` (although there are subtle differences between conventional Unix read/write semantics and those of OSS audio devices). It also supports a number of audio-specific methods; see below for the complete list of methods.

`device` is the audio device filename to use. If it is not specified, this module first looks in the environment variable `AUDIODEV` for a device to use. If not found, it falls back to `/dev/dsp`.

`mode` is one of `'r'` for read-only (record) access, `'w'` for write-only (playback) access and `'rw'` for both. Since many sound cards only allow one process to have the recorder or player open at a time, it is a good idea to open the device only for the activity needed. Further, some sound cards are half-duplex: they can be opened for reading or writing, but not both at once.

Note the unusual calling syntax: the *first* argument is optional, and the second is required. This is a historical artifact for compatibility with the older `linuxaudiodev` module which `ossaudiodev` supersedes.

ossaudiodev. `openmixer([device])`

Open a mixer device and return an OSS mixer device object. `device` is the mixer device filename to use. If it is not specified, this module first looks in the environment variable `MIXERDEV` for a device to use. If not found, it falls back to `/dev/mixer`. 
Before you can write to or read from an audio device, you must call three methods in the correct order:

1. `setfmt()` to set the output format
2. `channels()` to set the number of channels
3. `speed()` to set the sample rate

Alternately, you can use the `setparameters()` method to set all three audio parameters at once. This is more convenient, but may not be as flexible in all cases.

The audio device objects returned by `open()` define the following methods and (read-only) attributes:

- **`oss_audio_device.close()`**
  - Explicitly close the audio device. When you are done writing to or reading from an audio device, you should explicitly close it. A closed device cannot be used again.

- **`oss_audio_device.fileno()`**
  - Return the file descriptor associated with the device.

- **`oss_audio_device.read(size)`**
  - Read `size` bytes from the audio input and return them as a Python string. Unlike most Unix device drivers, OSS audio devices in blocking mode (the default) will block `read()` until the entire requested amount of data is available.

- **`oss_audio_device.write(data)`**
  - Write the Python string `data` to the audio device and return the number of bytes written. If the audio device is in blocking mode (the default), the entire string is always written (again, this is
different from usual Unix device semantics). If the device is in non-blocking mode, some data may not be written — see writeall().

oss_audio_device.writeall(data)
Write the entire Python string *data* to the audio device: waits until the audio device is able to accept data, writes as much data as it will accept, and repeats until *data* has been completely written. If the device is in blocking mode (the default), this has the same effect as write(); writeall() is only useful in non-blocking mode. Has no return value, since the amount of data written is always equal to the amount of data supplied.

The following methods each map to exactly one ioctl() system call. The correspondence is obvious: for example, setfmt() corresponds to the SNDCTL_DSP_SETFMT ioctl, and sync() to SNDCTL_DSP_SYNC (this can be useful when consulting the OSS documentation). If the underlying ioctl() fails, they all raise IOError.

oss_audio_device.nonblock()
Put the device into non-blocking mode. Once in non-blocking mode, there is no way to return it to blocking mode.

oss_audio_device.getfmts()
Return a bitmask of the audio output formats supported by the soundcard. Some of the formats supported by OSS are:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFMT_MU_LAW</td>
<td>a logarithmic encoding (used by Sun .au files and /dev/audio)</td>
</tr>
<tr>
<td>AFMT_A_LAW</td>
<td>a logarithmic encoding</td>
</tr>
<tr>
<td>AFMTIMA_ADPCM</td>
<td>a 4:1 compressed format defined by the Interactive Multimedia Association</td>
</tr>
<tr>
<td>AFMT_U8</td>
<td>Unsigned, 8-bit audio</td>
</tr>
<tr>
<td>Format</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>AFMT_S16_LE</td>
<td>Signed, 16-bit audio, little-endian byte order (as used by Intel processors)</td>
</tr>
<tr>
<td>AFMT_S16_BE</td>
<td>Signed, 16-bit audio, big-endian byte order (as used by 68k, PowerPC, Sparc)</td>
</tr>
<tr>
<td>AFMT_S8</td>
<td>Signed, 8 bit audio</td>
</tr>
<tr>
<td>AFMT_U16_LE</td>
<td>Unsigned, 16-bit little-endian audio</td>
</tr>
<tr>
<td>AFMT_U16_BE</td>
<td>Unsigned, 16-bit big-endian audio</td>
</tr>
</tbody>
</table>

Consult the OSS documentation for a full list of audio formats, and note that most devices support only a subset of these formats. Some older devices only support AFMT_U8; the most common format used today is AFMT_S16_LE.

oss_audio_device. **setfmt**(format)

Try to set the current audio format to format—see getfmts() for a list. Returns the audio format that the device was set to, which may not be the requested format. May also be used to return the current audio format—do this by passing an “audio format” of AFMT_QUERY.

oss_audio_device. **channels**(nchannels)

Set the number of output channels to nchannels. A value of 1 indicates monophonic sound, 2 stereophonic. Some devices may have more than 2 channels, and some high-end devices may not support mono. Returns the number of channels the device was set to.

oss_audio_device. **speed**(samplerate)

Try to set the audio sampling rate to samplerate samples per second. Returns the rate actually set. Most sound devices don’t support arbitrary sampling rates. Common rates are:

<table>
<thead>
<tr>
<th>Rate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8000</td>
<td>default rate for /dev/audio</td>
</tr>
<tr>
<td>11025</td>
<td>speech recording</td>
</tr>
</tbody>
</table>
oss_audio_device. **sync()**

Wait until the sound device has played every byte in its buffer. (This happens implicitly when the device is closed.) The OSS documentation recommends closing and re-opening the device rather than using **sync()**.

oss_audio_device. **reset()**

Immediately stop playing or recording and return the device to a state where it can accept commands. The OSS documentation recommends closing and re-opening the device after calling **reset()**.

oss_audio_device. **post()**

Tell the driver that there is likely to be a pause in the output, making it possible for the device to handle the pause more intelligently. You might use this after playing a spot sound effect, before waiting for user input, or before doing disk I/O.

The following convenience methods combine several ioctl's, or one ioctl and some simple calculations.

oss_audio_device. **setparameters**(format, nchannels, samplerate[, strict=False])

Set the key audio sampling parameters—sample format, number of channels, and sampling rate—in one method call. *format, nchannels, and samplerate* should be as specified in the **setfmt(), channels(), and speed()** methods. If *strict* is true, **setparameters()** checks to see if each parameter was actually set to the requested value, and raises **OSSAudioError** if not.
Returns a tuple \((\text{format}, \ n\text{channels}, \ \text{samplerate})\) indicating the parameter values that were actually set by the device driver (i.e., the same as the return values of \texttt{setfmt()}, \texttt{channels()}, \texttt{and speed()}).

For example,

\[
(fmt, \ channels, \ rate) = \text{dsp.setparameters(fmt, channels, rate)}
\]

is equivalent to

\[
\begin{align*}
\text{fmt} &= \text{dsp.setfmt(fmt)} \\
\text{channels} &= \text{dsp.channels(channels)} \\
\text{rate} &= \text{dsp.rate(rate)}
\end{align*}
\]

\texttt{oss\_audio\_device.bufsize()}

Returns the size of the hardware buffer, in samples.

\texttt{oss\_audio\_device.obufcount()}

Returns the number of samples that are in the hardware buffer yet to be played.

\texttt{oss\_audio\_device.obuffree()}

Returns the number of samples that could be queued into the hardware buffer to be played without blocking.

Audio device objects also support several read-only attributes:

\texttt{oss\_audio\_device.closed}

Boolean indicating whether the device has been closed.

\texttt{oss\_audio\_device.name}

String containing the name of the device file.

\texttt{oss\_audio\_device.mode}

The I/O mode for the file, either \"r\", \"rw\", or \"w\".
21.10.2. Mixer Device Objects

The mixer object provides two file-like methods:

```python
class oss_mixer_device:
    def close(self):
        This method closes the open mixer device file. Any further
        attempts to use the mixer after this file is closed will raise an
        IOError.

    def fileno(self):
        Returns the file handle number of the open mixer device file.
```

The remaining methods are specific to audio mixing:

```python
class oss_mixer_device:
    def controls(self):
        This method returns a bitmask specifying the available mixer
        controls ("Control" being a specific mixable "channel", such as
        SOUND_MIXER_PCM or SOUND_MIXER_SYNTH). This bitmask indicates a
        subset of all available mixer controls—the SOUND_MIXER_*
        constants defined at module level. To determine if, for example,
        the current mixer object supports a PCM mixer, use the following
        Python code:

        ```python
        mixer = ossaudiodev.openmixer()
        if mixer.controls() & (1 << ossaudiodev.SOUND_MIXER_PCM):
            # PCM is supported
            ... code ...
        ```

        For most purposes, the SOUND_MIXER_VOLUME (master volume) and
        SOUND_MIXER_PCM controls should suffice—but code that uses the
        mixer should be flexible when it comes to choosing mixer
        controls. On the Gravis Ultrasound, for example, SOUND_MIXER_VOLUME
        does not exist.
oss_mixer_device. **stereocontrols()**

Returns a bitmask indicating stereo mixer controls. If a bit is set, the corresponding control is stereo; if it is unset, the control is either monophonic or not supported by the mixer (use in combination with `controls()` to determine which).

See the code example for the `controls()` function for an example of getting data from a bitmask.

oss_mixer_device. **reccontrols()**

Returns a bitmask specifying the mixer controls that may be used to record. See the code example for `controls()` for an example of reading from a bitmask.

oss_mixer_device. **get(control)**

Returns the volume of a given mixer control. The returned volume is a 2-tuple `(left_volume, right_volume)`. Volumes are specified as numbers from 0 (silent) to 100 (full volume). If the control is monophonic, a 2-tuple is still returned, but both volumes are the same.

Raises **OSSAudioError** if an invalid control was specified, or **IOError** if an unsupported control is specified.

oss_mixer_device. **set(control, (left, right))**

Sets the volume for a given mixer control to `(left, right)`. `left` and `right` must be ints and between 0 (silent) and 100 (full volume). On success, the new volume is returned as a 2-tuple. Note that this may not be exactly the same as the volume specified, because of the limited resolution of some soundcard's mixers.

Raises **OSSAudioError** if an invalid mixer control was specified, or if the specified volumes were out-of-range.
oss_mixer_device.get_recsrc()

This method returns a bitmask indicating which control(s) are currently being used as a recording source.

oss_mixer_device.set_recsrc(bitmask)

Call this function to specify a recording source. Returns a bitmask indicating the new recording source (or sources) if successful; raises IOError if an invalid source was specified. To set the current recording source to the microphone input:

```python
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```
22. Internationalization

The modules described in this chapter help you write software that is independent of language and locale by providing mechanisms for selecting a language to be used in program messages or by tailoring output to match local conventions.

The list of modules described in this chapter is:

- **22.1. gettext — Multilingual internationalization services**
  - **22.1.1. GNU gettext API**
  - **22.1.2. Class-based API**
    - **22.1.2.1. The NullTranslations class**
    - **22.1.2.2. The GNUTranslations class**
    - **22.1.2.3. Solaris message catalog support**
    - **22.1.2.4. The Catalog constructor**
  - **22.1.3. Internationalizing your programs and modules**
    - **22.1.3.1. Localizing your module**
    - **22.1.3.2. Localizing your application**
    - **22.1.3.3. Changing languages on the fly**
    - **22.1.3.4. Deferred translations**
    - **22.1.3.5. gettext() vs. lgettext()**
  - **22.1.4. Acknowledgements**
- **22.2. locale — Internationalization services**
  - **22.2.1. Background, details, hints, tips and caveats**
  - **22.2.2. For extension writers and programs that embed Python**
  - **22.2.3. Access to message catalogs**
The `gettext` module provides internationalization (I18N) and localization (L10N) services for your Python modules and applications. It supports both the GNU `gettext` message catalog API and a higher level, class-based API that may be more appropriate for Python files. The interface described below allows you to write your module and application messages in one natural language, and provide a catalog of translated messages for running under different natural languages.

Some hints on localizing your Python modules and applications are also given.
22.1.1. GNU gettext API

The `gettext` module defines the following API, which is very similar to the GNU `gettext` API. If you use this API you will affect the translation of your entire application globally. Often this is what you want if your application is monolingual, with the choice of language dependent on the locale of your user. If you are localizing a Python module, or if your application needs to switch languages on the fly, you probably want to use the class-based API instead.

`gettext.bindtextdomain(domain[, localedir])`

Bind the `domain` to the locale directory `localedir`. More concretely, `gettext` will look for binary `.mo` files for the given domain using the path (on Unix): `localedir/language/LC_MESSAGES/domain.mo`, where `languages` is searched for in the environment variables `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG` respectively.

If `localedir` is omitted or `None`, then the current binding for `domain` is returned. [1]

`gettext.bind_textdomain_codeset(domain[, codeset])`

Bind the `domain` to `codeset`, changing the encoding of strings returned by the `gettext()` family of functions. If `codeset` is omitted, then the current binding is returned.

*New in version 2.4.*

`gettext.textdomain([domain])`

Change or query the current global domain. If `domain` is `None`, then the current global domain is returned, otherwise the global domain is set to `domain`, which is returned.
gettext. **gettext**(message)
Return the localized translation of message, based on the current global domain, language, and locale directory. This function is usually aliased as \_(\) in the local namespace (see examples below).

gettext. **lgettext**(message)
Equivalent to gettext(), but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with bind_textdomain_codeset().

*New in version 2.4.*

gettext. **dgettext**(domain, message)
Like gettext(), but look the message up in the specified domain.

gettext. **ldgettext**(domain, message)
Equivalent to dgettext(), but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with bind_textdomain_codeset().

*New in version 2.4.*

gettext. **ngettext**(singular, plural, n)
Like gettext(), but consider plural forms. If a translation is found, apply the plural formula to n, and return the resulting message (some languages have more than two plural forms). If no translation is found, return singular if n is 1; return plural otherwise.

The Plural formula is taken from the catalog header. It is a C or Python expression that has a free variable n; the expression evaluates to the index of the plural in the catalog. See the GNU gettext documentation for the precise syntax to be used in .po files and the formulas for a variety of languages.
New in version 2.3.

`gettext.lngettext(singular, plural, n)`

Equivalent to `ngettext()`, but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with `bind_textdomain_codeset()`.

New in version 2.4.

`gettext.dngettext(domain, singular, plural, n)`

Like `ngettext()`, but look the message up in the specified domain.

New in version 2.3.

`gettext.ldngettext(domain, singular, plural, n)`

Equivalent to `dngettext()`, but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with `bind_textdomain_codeset()`.

New in version 2.4.

Note that GNU `gettext` also defines a `dcgettext()` method, but this was deemed not useful and so it is currently unimplemented.

Here's an example of typical usage for this API:

```python
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
ggettext.textdomain('myapplication')
_
_
_
_
print _('This is a translatable string.')
```
22.1.2. Class-based API

The class-based API of the gettext module gives you more flexibility and greater convenience than the GNU gettext API. It is the recommended way of localizing your Python applications and modules. gettext defines a “translations” class which implements the parsing of GNU .mo format files, and has methods for returning either standard 8-bit strings or Unicode strings. Instances of this “translations” class can also install themselves in the built-in namespace as the function `_()`.

```
gettext.find([domain[, localedir[, languages[, all]]]])
```

This function implements the standard .mo file search algorithm. It takes a *domain*, identical to what `textdomain()` takes. Optional *localedir* is as in `bindtextdomain()` Optional *languages* is a list of strings, where each string is a language code.

If *localedir* is not given, then the default system locale directory is used. [2] If *languages* is not given, then the following environment variables are searched: `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG`. The first one returning a non-empty value is used for the *languages* variable. The environment variables should contain a colon separated list of languages, which will be split on the colon to produce the expected list of language code strings.

`find()` then expands and normalizes the languages, and then iterates through them, searching for an existing file built of these components:

```
localedir/language/LC_MESSAGES/domain.mo
```

The first such file name that exists is returned by `find()`. If no
such file is found, then None is returned. If all is given, it returns a list of all file names, in the order in which they appear in the languages list or the environment variables.

```
gettext.translation(domain[, localedir[, languages[, class_[, fallback[, codeset]]]]]]
```

Return a Translations instance based on the domain, localedir, and languages, which are first passed to find() to get a list of the associated .mo file paths. Instances with identical .mo file names are cached. The actual class instantiated is either class_ if provided, otherwise GNUTranslations. The class's constructor must take a single file object argument. If provided, codeset will change the charset used to encode translated strings.

If multiple files are found, later files are used as fallbacks for earlier ones. To allow setting the fallback, copy.copy() is used to clone each translation object from the cache; the actual instance data is still shared with the cache.

If no .mo file is found, this function raises IOError if fallback is false (which is the default), and returns a NullTranslations instance if fallback is true.

*Changed in version 2.4:* Added the codeset parameter.

```
gettext.install(domain[, localedir[, unicode[, codeset[, names]]]]]
```

This installs the function _() in Python’s builtins namespace, based on domain, localedir, and codeset which are passed to the function translation(). The unicode flag is passed to the resulting translation object’s install() method.

For the names parameter, please see the description of the
As seen below, you usually mark the strings in your application that are candidates for translation, by wrapping them in a call to the \_() function, like this:

```python
print \_('This string will be translated.\')
```

For convenience, you want the \_() function to be installed in Python’s builtins namespace, so it is easily accessible in all modules of your application.

*Changed in version 2.4:* Added the codeset parameter.

*Changed in version 2.5:* Added the names parameter.

### 22.1.2.1. The NullTranslations class

Translation classes are what actually implement the translation of original source file message strings to translated message strings. The base class used by all translation classes is `NullTranslations`; this provides the basic interface you can use to write your own specialized translation classes. Here are the methods of `NullTranslations`:

```python
class gettext.NullTranslations([fp])
```

Takes an optional file object `fp`, which is ignored by the base class. Initializes “protected” instance variables `\_info` and `\_charset` which are set by derived classes, as well as `\_fallback`, which is set through `add_fallback()`. It then calls `self.\_parse(fp)` if `fp` is not `None`.

```python
\_parse(fp)
```

No-op’d in the base class, this method takes file object `fp`, and
reads the data from the file, initializing its message catalog. If you have an unsupported message catalog file format, you should override this method to parse your format.

**add_fallback(fallback)**
Add *fallback* as the fallback object for the current translation object. A translation object should consult the fallback if it cannot provide a translation for a given message.

**gettext(message)**
If a fallback has been set, forward `gettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

**lgettext(message)**
If a fallback has been set, forward `lgettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

*New in version 2.4.*

**ugettext(message)**
If a fallback has been set, forward `ugettext()` to the fallback. Otherwise, return the translated message as a Unicode string. Overridden in derived classes.

**ngettext(singular, plural, n)**
If a fallback has been set, forward `ngettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

*New in version 2.3.*

**lngettext(singular, plural, n)**
If a fallback has been set, forward `lngettext()` to the fallback.
Otherwise, return the translated message. Overridden in derived classes.

*New in version 2.4.*

ungettext(*singular, plural, n*)
If a fallback has been set, forward ungettext() to the fallback. Otherwise, return the translated message as a Unicode string. Overridden in derived classes.

*New in version 2.3.*

info()
Return the “protected” _info variable.

charset()
Return the “protected” _charset variable.

output_charset()
Return the “protected” _output_charset variable, which defines the encoding used to return translated messages.

*New in version 2.4.*

set_output_charset(*charset*)
Change the “protected” _output_charset variable, which defines the encoding used to return translated messages.

*New in version 2.4.*

install([*unicode*, *names]*)
If the unicode flag is false, this method installs self.gettext() into the built-in namespace, binding it to _. If unicode is true, it binds self.ugettext() instead. By default, unicode is false.
If the *names* parameter is given, it must be a sequence containing the names of functions you want to install in the builtins namespace in addition to _(). Supported names are 'gettext' (bound to `self.gettext()` or `self.ugettext()` according to the *unicode* flag), 'ngettext' (bound to `self.ngettext()` or `self.ungettext()` according to the *unicode* flag), 'lgettext' and 'lngettext'.

Note that this is only one way, albeit the most convenient way, to make the _() function available to your application. Because it affects the entire application globally, and specifically the built-in namespace, localized modules should never install _(). Instead, they should use this code to make _() available to their module:

```python
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

This puts _() only in the module’s global namespace and so only affects calls within this module.

*Changed in version 2.5:* Added the *names* parameter.

### 22.1.2.2. The **GNUTranslations** class

The gettext module provides one additional class derived from `NullTranslations`: `GNUTranslations`. This class overrides `_parse()` to enable reading GNU gettext format .mo files in both big-endian and little-endian format. It also coerces both message ids and message strings to Unicode.

`GNUTranslations` parses optional meta-data out of the translation catalog. It is convention with GNU gettext to include meta-data as
the translation for the empty string. This meta-data is in RFC 822-style key: value pairs, and should contain the Project-Id-Version key. If the key Content-Type is found, then the charset property is used to initialize the “protected” _charset instance variable, defaulting to None if not found. If the charset encoding is specified, then all message ids and message strings read from the catalog are converted to Unicode using this encoding. The ugettext() method always returns a Unicode, while the gettext() returns an encoded 8-bit string. For the message id arguments of both methods, either Unicode strings or 8-bit strings containing only US-ASCII characters are acceptable. Note that the Unicode version of the methods (i.e. ugettext() and ungettext()) are the recommended interface to use for internationalized Python programs.

The entire set of key/value pairs are placed into a dictionary and set as the “protected” _info instance variable.

If the .mo file’s magic number is invalid, or if other problems occur while reading the file, instantiating a GNUTranslations class can raise IOError.

The following methods are overridden from the base class implementation:

GNUTranslations.gettext(message)
Look up the message id in the catalog and return the corresponding message string, as an 8-bit string encoded with the catalog’s charset encoding, if known. If there is no entry in the catalog for the message id, and a fallback has been set, the look up is forwarded to the fallback’s gettext() method. Otherwise, the message id is returned.

GNUTranslations.lgettext(message)
Equivalent to gettext(), but the translation is returned in the
preferred system encoding, if no other encoding was explicitly set with `set_output_charset()`.

**New in version 2.4.**

**GNUTranslations.** `ugettext(message)`

Look up the `message` id in the catalog and return the corresponding message string, as a Unicode string. If there is no entry in the catalog for the `message` id, and a fallback has been set, the look up is forwarded to the fallback's `ugettext()` method. Otherwise, the `message` id is returned.

**GNUTranslations.** `ngettext(singular, plural, n)`

Do a plural-forms lookup of a message id. `singular` is used as the message id for purposes of lookup in the catalog, while `n` is used to determine which plural form to use. The returned message string is an 8-bit string encoded with the catalog's charset encoding, if known.

If the message id is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's `ngettext()` method. Otherwise, when `n` is 1 `singular` is returned, and `plural` is returned in all other cases.

**New in version 2.3.**

**GNUTranslations.** `lngettext(singular, plural, n)`

Equivalent to `gettext()`, but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with `set_output_charset()`.

**New in version 2.4.**

**GNUTranslations.** `ungettext(singular, plural, n)`

Do a plural-forms lookup of a message id. `singular` is used as the
message id for purposes of lookup in the catalog, while \( n \) is used to determine which plural form to use. The returned message string is a Unicode string.

If the message id is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's \texttt{ungettext()}
method. Otherwise, when \( n \) is 1 \textit{singular} is returned, and \textit{plural} is returned in all other cases.

Here is an example:

\begin{verbatim}
\begin{verbatim}
    n = len(os.listdir('.'))
    cat = GNUTranslations(somefile)
    message = cat.ungettext(
        'There is \%(num)d file in this directory',
        'There are \%(num)d files in this directory',
        n) % {'num': n}
\end{verbatim}
\end{verbatim}
\end{verbatim}

\textit{New in version 2.3.}

\section*{22.1.2.3. Solaris message catalog support}

The Solaris operating system defines its own binary \texttt{.mo} file format, but since no documentation can be found on this format, it is not supported at this time.

\section*{22.1.2.4. The Catalog constructor}

\begin{verbatim}
    import gettext
    cat = gettext.Catalog(domain, localedir)
    _ = cat.gettext
    print _(\textit{'hello world'})
\end{verbatim}

\begin{verbatim}
\end{verbatim}
For compatibility with this older module, the function `catalog()` is an alias for the `translation()` function described above.

One difference between this module and Henstridge’s: his catalog objects supported access through a mapping API, but this appears to be unused and so is not currently supported.
Internationalization (I18N) refers to the operation by which a program is made aware of multiple languages. Localization (L10N) refers to the adaptation of your program, once internationalized, to the local language and cultural habits. In order to provide multilingual messages for your Python programs, you need to take the following steps:

1. prepare your program or module by specially marking translatable strings
2. run a suite of tools over your marked files to generate raw messages catalogs
3. create language specific translations of the message catalogs
4. use the gettext module so that message strings are properly translated

In order to prepare your code for I18N, you need to look at all the strings in your files. Any string that needs to be translated should be marked by wrapping it in `_` — that is, a call to the function `._()`. For example:

```python
def example_function():
    filename = 'mylog.txt'
    message = _('writing a log message')
    fp = open(filename, 'w')
    fp.write(message)
    fp.close()
```

In this example, the string 'writing a log message' is marked as a candidate for translation, while the strings 'mylog.txt' and 'w' are not.

The Python distribution comes with two tools which help you
generate the message catalogs once you’ve prepared your source code. These may or may not be available from a binary distribution, but they can be found in a source distribution, in the Tools/i18n directory.

The `pygettext` [3] program scans all your Python source code looking for the strings you previously marked as translatable. It is similar to the GNU `gettext` program except that it understands all the intricacies of Python source code, but knows nothing about C or C++ source code. You don’t need GNU `gettext` unless you’re also going to be translating C code (such as C extension modules).

`pygettext` generates textual Uniforum-style human readable message catalog `.pot` files, essentially structured human readable files which contain every marked string in the source code, along with a placeholder for the translation strings. `pygettext` is a command line script that supports a similar command line interface as `xgettext`; for details on its use, run:

```
pygettext.py --help
```

Copies of these `.pot` files are then handed over to the individual human translators who write language-specific versions for every supported natural language. They send you back the filled in language-specific versions as a `.po` file. Using the `msgfmt.py` [4] program (in the Tools/i18n directory), you take the `.po` files from your translators and generate the machine-readable `.mo` binary catalog files. The `.mo` files are what the `gettext` module uses for the actual translation processing during run-time.

How you use the `gettext` module in your code depends on whether you are internationalizing a single module or your entire application. The next two sections will discuss each case.
22.1.3.1. Localizing your module

If you are localizing your module, you must take care not to make global changes, e.g. to the built-in namespace. You should not use the GNU gettext API but instead the class-based API.

Let’s say your module is called “spam” and the module’s various natural language translation .mo files reside in `/usr/share/locale` in GNU gettext format. Here’s what you would put at the top of your module:

```python
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.lgettext
```

If your translators were providing you with Unicode strings in their .po files, you’d instead do:

```python
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.ugettext
```

22.1.3.2. Localizing your application

If you are localizing your application, you can install the `_()` function globally into the built-in namespace, usually in the main driver file of your application. This will let all your application-specific files just use `_('...')` without having to explicitly install it in each file.

In the simple case then, you need only add the following bit of code to the main driver file of your application:

```python
import gettext
ggettext.install('myapplication')
```

If you need to set the locale directory or the `unicode` flag, you can
pass these into the `install()` function:

```python
import gettext
gettext.install('myapplication', '/usr/share/locale', unicode=1)
```

### 22.1.3.3. Changing languages on the fly

If your program needs to support many languages at the same time, you may want to create multiple translation instances and then switch between them explicitly, like so:

```python
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

### 22.1.3.4. Deferred translations

In most coding situations, strings are translated where they are coded. Occasionally however, you need to mark strings for translation, but defer actual translation until later. A classic example is:

```python
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python',
          ]

# ...
```
for a in animals:
    print a

Here, you want to mark the strings in the animals list as being translatable, but you don’t actually want to translate them until they are printed.

Here is one way you can handle this situation:

def _(message): return message

animals = [_("mollusk"),
           _("albatross"),
           _("rat"),
           _("penguin"),
           _("python"), ]

del _

# ...
for a in animals:
    print _(a)

This works because the dummy definition of _() simply returns the string unchanged. And this dummy definition will temporarily override any definition of _() in the built-in namespace (until the del command). Take care, though if you have a previous definition of _() in the local namespace.

Note that the second use of _() will not identify “a” as being translatable to the pygettext program, since it is not a string.

Another way to handle this is with the following example:

def N_(message): return message

animals = [N_("mollusk"),
           N_("albatross"),
           N_("rat"),
           N_("penguin"),]
In this case, you are marking translatable strings with the function `n_()`, [5] which won't conflict with any definition of `_()`. However, you will need to teach your message extraction program to look for translatable strings marked with `n_()`. `pygettext` and `xpot` both support this through the use of command line switches.

22.1.3.5. gettext() vs. lgettext()

In Python 2.4 the `lgettext()` family of functions were introduced. The intention of these functions is to provide an alternative which is more compliant with the current implementation of GNU gettext. Unlike `gettext()`, which returns strings encoded with the same codeset used in the translation file, `lgettext()` will return strings encoded with the preferred system encoding, as returned by `locale.getpreferredencoding()`. Also notice that Python 2.4 introduces new functions to explicitly choose the codeset used in translated strings. If a codeset is explicitly set, even `lgettext()` will return translated strings in the requested codeset, as would be expected in the GNU gettext implementation.
22.1.4. Acknowledgements

The following people contributed code, feedback, design suggestions, previous implementations, and valuable experience to the creation of this module:

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw
- Gustavo Niemeyer

Footnotes

[1] The default locale directory is system dependent; for example, on RedHat Linux it is /usr/share/locale, but on Solaris it is /usr/lib/locale. The gettext module does not try to support these system dependent defaults; instead its default is sys.prefix/share/locale. For this reason, it is always best to call bindtextdomain() with an explicit absolute path at the start of your application.

[2] See the footnote for bindtextdomain() above.

[3] François Pinard has written a program called xpot which does a similar job. It is available as part of his po-utils package. msgfmt.py is binary compatible with GNU msgfmt except that it provides a simpler, all-Python implementation. With this and pygettext.py, you generally won’t need to install the GNU gettext package to internationalize your Python applications.

[4] The choice of N_() here is totally arbitrary; it could have just as easily been MarkThisStringForTranslation().
22.2. locale — Internationalization services

The `locale` module opens access to the POSIX locale database and functionality. The POSIX locale mechanism allows programmers to deal with certain cultural issues in an application, without requiring the programmer to know all the specifics of each country where the software is executed.

The `locale` module is implemented on top of the `_locale` module, which in turn uses an ANSI C locale implementation if available.

The `locale` module defines the following exception and functions:

```plaintext
exception locale.Error
    Exception raised when the locale passed to `setlocale()` is not recognized.

locale.setlocale(category[, locale])
    If `locale` is given and not `None`, `setlocale()` modifies the locale setting for the `category`. The available categories are listed in the data description below. `locale` may be a string, or an iterable of two strings (language code and encoding). If it's an iterable, it's converted to a locale name using the locale aliasing engine. An empty string specifies the user’s default settings. If the modification of the locale fails, the exception `Error` is raised. If successful, the new locale setting is returned.

    If `locale` is omitted or `None`, the current setting for `category` is returned.

setlocale() is not thread-safe on most systems. Applications
typically start with a call of

```python
import locale
locale.setlocale(locale.LC_ALL, '')
```

This sets the locale for all categories to the user’s default setting (typically specified in the `LANG` environment variable). If the locale is not changed thereafter, using multithreading should not cause problems.

*Changed in version 2.0:* Added support for iterable values of the `locale` parameter.

```python
locale.localeconv()
```

Returns the database of the local conventions as a dictionary. This dictionary has the following strings as keys:

<table>
<thead>
<tr>
<th>Category</th>
<th>Key</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LC_NUMERIC</strong></td>
<td>'decimal_point'</td>
<td>Decimal point character.</td>
</tr>
<tr>
<td></td>
<td>'grouping'</td>
<td>Sequence of numbers specifying which relative positions the 'thousands_sep' is expected. If the sequence is terminated with <code>CHAR_MAX</code>, no further grouping is performed. If the sequence terminates with a 0, the last group size is repeatedly</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>'thousands_sep'</td>
<td>Character used between groups.</td>
<td></td>
</tr>
<tr>
<td><strong>LC_MONETARY</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'int_curr_symbol'</td>
<td>International currency symbol.</td>
<td></td>
</tr>
<tr>
<td>'currency_symbol'</td>
<td>Local currency symbol.</td>
<td></td>
</tr>
<tr>
<td>'p_cs_precedes/n_cs_precedes'</td>
<td>Whether the currency symbol precedes the value (for positive resp. negative values).</td>
<td></td>
</tr>
<tr>
<td>'p_sep_by_space/n_sep_by_space'</td>
<td>Whether the currency symbol is separated from the value by a space (for positive resp. negative values).</td>
<td></td>
</tr>
<tr>
<td>'mon_decimal_point'</td>
<td>Decimal point used for monetary values.</td>
<td></td>
</tr>
<tr>
<td>'frac_digits'</td>
<td>Number of fractional digits used in local formatting of monetary values.</td>
<td></td>
</tr>
<tr>
<td>'int_frac_digits'</td>
<td>international formatting of monetary values.</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>'mon_thousands_sep'</td>
<td>Group separator used for monetary values.</td>
<td></td>
</tr>
<tr>
<td>'mon_grouping'</td>
<td>Equivalent to 'grouping', used for monetary values.</td>
<td></td>
</tr>
<tr>
<td>'positive_sign'</td>
<td>Symbol used to annotate a positive monetary value.</td>
<td></td>
</tr>
<tr>
<td>'negative_sign'</td>
<td>Symbol used to annotate a negative monetary value.</td>
<td></td>
</tr>
<tr>
<td>'p_sign_posn/n_sign_posn'</td>
<td>The position of the sign (for positive resp. negative values), see below.</td>
<td></td>
</tr>
</tbody>
</table>

All numeric values can be set to **CHAR_MAX** to indicate that there is no value specified in this locale.

The possible values for 'p_sign_posn' and 'n_sign_posn' are given below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Currency and value are surrounded by parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>The sign should precede the value and currency symbol.</td>
</tr>
<tr>
<td>2</td>
<td>The sign should follow the value and currency symbol.</td>
</tr>
<tr>
<td>3</td>
<td>The sign should immediately precede the value.</td>
</tr>
<tr>
<td>4</td>
<td>The sign should immediately follow the value.</td>
</tr>
<tr>
<td>CHAR_MAX</td>
<td>Nothing is specified in this locale.</td>
</tr>
</tbody>
</table>

**locale.nl_langinfo(option)**

Return some locale-specific information as a string. This function is not available on all systems, and the set of possible options might also vary across platforms. The possible argument values are numbers, for which symbolic constants are available in the `locale` module.

The `nl_langinfo()` function accepts one of the following keys. Most descriptions are taken from the corresponding description in the GNU C library.

**locale.CODESET**

Get a string with the name of the character encoding used in the selected locale.

**locale.D_T_FMT**

Get a string that can be used as a format string for `time.strftime()` to represent date and time in a locale-specific way.

**locale.D_FMT**

Get a string that can be used as a format string for `time.strftime()` to represent a date in a locale-specific way.

**locale.T_FMT**

Get a string that can be used as a format string for
*time.strftime()* to represent a time in a locale-specific way.

**locale.T_FMT_AMPM**
Get a format string for *time.strftime()* to represent time in the am/pm format.

**DAY_1 ... DAY_7**
Get the name of the n-th day of the week.

**Note:** This follows the US convention of **DAY_1** being Sunday, not the international convention (ISO 8601) that Monday is the first day of the week.

**ABDAY_1 ... ABDAY_7**
Get the abbreviated name of the n-th day of the week.

**MON_1 ... MON_12**
Get the name of the n-th month.

**ABMON_1 ... ABMON_12**
Get the abbreviated name of the n-th month.

**locale.RADIXCHAR**
Get the radix character (decimal dot, decimal comma, etc.).

**locale.THOUSEP**
Get the separator character for thousands (groups of three digits).

**locale.YESEXPR**
Get a regular expression that can be used with the regex function to recognize a positive response to a yes/no question.

**Note:** The expression is in the syntax suitable for the
regex() function from the C library, which might differ from the syntax used in re.

locale. **NOEXPR**

Get a regular expression that can be used with the regex(3) function to recognize a negative response to a yes/no question.

locale. **CRNCYSTR**

Get the currency symbol, preceded by “-” if the symbol should appear before the value, “+” if the symbol should appear after the value, or “.” if the symbol should replace the radix character.

locale. **ERA**

Get a string that represents the era used in the current locale.

Most locales do not define this value. An example of a locale which does define this value is the Japanese one. In Japan, the traditional representation of dates includes the name of the era corresponding to the then-emperor's reign.

Normally it should not be necessary to use this value directly. Specifying the **E** modifier in their format strings causes the time.strftime() function to use this information. The format of the returned string is not specified, and therefore you should not assume knowledge of it on different systems.

locale. **ERA_D_T_FMT**

Get a format string for time.strftime() to represent date and time in a locale-specific era-based way.

locale. **ERA_D_FMT**

Get a format string for time.strftime() to represent a date in
a locale-specific era-based way.

locale.ERA_T_FMT

Get a format string for time.strftime() to represent a time in a locale-specific era-based way.

locale.ALT_DIGITS

Get a representation of up to 100 values used to represent the values 0 to 99.

locale.getdefaultlocale([envvars])

Tries to determine the default locale settings and returns them as a tuple of the form (language code, encoding).

According to POSIX, a program which has not called setlocale(LC_ALL, '') runs using the portable 'C' locale. Calling setlocale(LC_ALL, '') lets it use the default locale as defined by the LANG variable. Since we do not want to interfere with the current locale setting we thus emulate the behavior in the way described above.

To maintain compatibility with other platforms, not only the LANG variable is tested, but a list of variables given as envvars parameter. The first found to be defined will be used. envvars defaults to the search path used in GNU gettext; it must always contain the variable name LANG. The GNU gettext search path contains 'LANGUAGE', 'LC_ALL', 'LC_CTYPE', and 'LANG', in that order.

Except for the code 'C', the language code corresponds to RFC 1766. language code and encoding may be None if their values cannot be determined.

New in version 2.0.
locale.getlocale([category])

Returns the current setting for the given locale category as sequence containing language code, encoding. category may be one of the LC_* values except LC_ALL. It defaults to LC_CTYPE.

Except for the code 'c', the language code corresponds to RFC 1766. language code and encoding may be None if their values cannot be determined.

New in version 2.0.

locale.getpreferredencoding([do_setlocale])

Return the encoding used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess.

On some systems, it is necessary to invoke setlocale() to obtain the user preferences, so this function is not thread-safe. If invoking setlocale is not necessary or desired, do_setlocale should be set to False.

New in version 2.3.

locale.normalize(localename)

Returns a normalized locale code for the given locale name. The returned locale code is formatted for use with setlocale(). If normalization fails, the original name is returned unchanged.

If the given encoding is not known, the function defaults to the default encoding for the locale code just like setlocale().

New in version 2.0.
locale.**resetlocale**(category)

Sets the locale for category to the default setting.

The default setting is determined by calling **getdefaultlocale()**. **category** defaults to **LC_ALL**.

*New in version 2.0.*

locale.**strcoll**(string1, string2)

Compares two strings according to the current **LC_COLLATE** setting. As any other compare function, returns a negative, or a positive value, or 0, depending on whether **string1** collates before or after **string2** or is equal to it.

locale.**strxfrm**(string)

Transforms a string to one that can be used for the built-in function **cmp()**, and still returns locale-aware results. This function can be used when the same string is compared repeatedly, e.g. when collating a sequence of strings.

locale.**format**(format, val[, grouping[, monetary]])

Formats a number **val** according to the current **LC_NUMERIC** setting. The format follows the conventions of the % operator. For floating point values, the decimal point is modified if appropriate. If **grouping** is true, also takes the grouping into account.

If **monetary** is true, the conversion uses monetary thousands separator and grouping strings.

Please note that this function will only work for exactly one %char specifier. For whole format strings, use **format_string()**.

*Changed in version 2.5:* Added the **monetary** parameter.
locale. `format_string(format, val[, grouping])`
Processes formatting specifiers as in `format % val`, but takes the current locale settings into account.

*New in version 2.5.*

locale. `currency(val[, symbol[, grouping[, international]]]])`
Formats a number `val` according to the current `LC_MONETARY` settings.

The returned string includes the currency symbol if `symbol` is true, which is the default. If `grouping` is true (which is not the default), grouping is done with the value. If `international` is true (which is not the default), the international currency symbol is used.

Note that this function will not work with the ‘C’ locale, so you have to set a locale via `setlocale()` first.

*New in version 2.5.*

locale. `str(float)`
Formats a floating point number using the same format as the built-in function `str(float)`, but takes the decimal point into account.

locale. `atof(string)`
Converts a string to a floating point number, following the `LC_NUMERIC` settings.

locale. `atoi(string)`
Converts a string to an integer, following the `LC_NUMERIC` conventions.

locale. `LC_CTYPE`
Locale category for the character type functions. Depending on
the settings of this category, the functions of module `string`
dealing with case change their behaviour.

`locale.LC_COLLATE`
Locale category for sorting strings. The functions `strcoll()` and
`strxfrm()` of the `locale` module are affected.

`locale.LC_TIME`
Locale category for the formatting of time. The function
`time.strftime()` follows these conventions.

`locale.LC_MONETARY`
Locale category for formatting of monetary values. The available
options are available from the `localeconv()` function.

`locale.LC_MESSAGES`
Locale category for message display. Python currently does not
support application specific locale-aware messages. Messages
displayed by the operating system, like those returned by
`os.strerror()` might be affected by this category.

`locale.LC_NUMERIC`
Locale category for formatting numbers. The functions `format()`,
`atoi()`, `atof()` and `str()` of the `locale` module are affected by
that category. All other numeric formatting operations are not
affected.

`locale.LC_ALL`
Combination of all locale settings. If this flag is used when the
locale is changed, setting the locale for all categories is
attempted. If that fails for any category, no category is changed at
all. When the locale is retrieved using this flag, a string indicating
the setting for all categories is returned. This string can be later
used to restore the settings.
locale.CHAR_MAX

This is a symbolic constant used for different values returned by localeconv().

Example:

```python
>>> import locale
>>> loc = locale.getlocale()  # get current locale
# use German locale; name might vary with platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('f\xe4n', 'foo')  # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '')  # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C')  # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc)  # restore saved locale
```
22.2.1. Background, details, hints, tips and caveats

The C standard defines the locale as a program-wide property that may be relatively expensive to change. On top of that, some implementation are broken in such a way that frequent locale changes may cause core dumps. This makes the locale somewhat painful to use correctly.

Initially, when a program is started, the locale is the C locale, no matter what the user’s preferred locale is. The program must explicitly say that it wants the user’s preferred locale settings by calling `setlocale(LC_ALL, '')`.

It is generally a bad idea to call `setlocale()` in some library routine, since as a side effect it affects the entire program. Saving and restoring it is almost as bad: it is expensive and affects other threads that happen to run before the settings have been restored.

If, when coding a module for general use, you need a locale independent version of an operation that is affected by the locale (such as `string.lower()`, or certain formats used with `time.strftime()`), you will have to find a way to do it without using the standard library routine. Even better is convincing yourself that using locale settings is okay. Only as a last resort should you document that your module is not compatible with non-C locale settings.

The case conversion functions in the `string` module are affected by the locale settings. When a call to the `setlocale()` function changes the `LC_CTYPE` settings, the variables `string.lowercase`, `string.uppercase` and `string.letters` are recalculated. Note that
code that uses these variable through ‘from ... import ...’, e.g. from string import letters, is not affected by subsequent setlocale() calls.

The only way to perform numeric operations according to the locale is to use the special functions defined by this module: atof(), atoi(), format(), str().
22.2.2. For extension writers and programs that embed Python

Extension modules should never call `setlocale()`, except to find out what the current locale is. But since the return value can only be used portably to restore it, that is not very useful (except perhaps to find out whether or not the locale is C).

When Python code uses the `locale` module to change the locale, this also affects the embedding application. If the embedding application doesn't want this to happen, it should remove the `_locale` extension module (which does all the work) from the table of built-in modules in the `config.c` file, and make sure that the `_locale` module is not accessible as a shared library.
22.2.3. Access to message catalogs

The locale module exposes the C library's gettext interface on systems that provide this interface. It consists of the functions gettext(), dgettext(), dcgettext(), textdomain(), bindtextdomain(), and bind_textdomain_codeset(). These are similar to the same functions in the gettext module, but use the C library's binary format for message catalogs, and the C library's search algorithms for locating message catalogs.

Python applications should normally find no need to invoke these functions, and should use gettext instead. A known exception to this rule are applications that link with additional C libraries which internally invoke gettext() or dcgettext(). For these applications, it may be necessary to bind the text domain, so that the libraries can properly locate their message catalogs.
23. Program Frameworks

The modules described in this chapter are frameworks that will largely dictate the structure of your program. Currently the modules described here are all oriented toward writing command-line interfaces.

The full list of modules described in this chapter is:

- **23.1. cmd — Support for line-oriented command interpreters**
  - 23.1.1. Cmd Objects
- **23.2. shlex — Simple lexical analysis**
  - 23.2.1. shlex Objects
  - 23.2.2. Parsing Rules
23.1. cmd — Support for line-oriented command interpreters

**Source code:** Lib/cmd.py

The `cmd` class provides a simple framework for writing line-oriented
command interpreters. These are often useful for test harnesses,
administrative tools, and prototypes that will later be wrapped in a
more sophisticated interface.

```python
class cmd.Cmd([completekey[, stdin[, stdout]]])
```

A `Cmd` instance or subclass instance is a line-oriented interpreter
framework. There is no good reason to instantiate `Cmd` itself;
rather, it’s useful as a superclass of an interpreter class you
define yourself in order to inherit `Cmd`’s methods and encapsulate
action methods.

The optional argument `completekey` is the `readline` name of a
completion key; it defaults to `Tab`. If `completekey` is not `None` and
`readline` is available, command completion is done automatically.

The optional arguments `stdin` and `stdout` specify the input and
output file objects that the `Cmd` instance or subclass instance will
use for input and output. If not specified, they will default to
`sys.stdin` and `sys.stdout`.

If you want a given `stdin` to be used, make sure to set the
instance’s `use_rawinput` attribute to `False`, otherwise `stdin` will be
ignored.
Changed in version 2.3: The stdin and stdout parameters were added.
23.1.1. Cmd Objects

A `Cmd` instance has the following methods:

```python
Cmd.cmdloop([intro])
```
Repeatedly issue a prompt, accept input, parse an initial prefix off the received input, and dispatch to action methods, passing them the remainder of the line as argument.

The optional argument is a banner or intro string to be issued before the first prompt (this overrides the `intro` class attribute).

If the `readline` module is loaded, input will automatically inherit `bash`-like history-list editing (e.g. `Control-P` scrolls back to the last command, `Control-N` forward to the next one, `Control-F` moves the cursor to the right non-destructively, `Control-B` moves the cursor to the left non-destructively, etc.).

An end-of-file on input is passed back as the string `'EOF'`.

An interpreter instance will recognize a command name `foo` if and only if it has a method `do_foo()`. As a special case, a line beginning with the character `?' is dispatched to the method `do_help()`. As another special case, a line beginning with the character `!' is dispatched to the method `do_shell()` (if such a method is defined).

This method will return when the `postcmd()` method returns a true value. The `stop` argument to `postcmd()` is the return value from the command’s corresponding `do_*()` method.

If completion is enabled, completing commands will be done
automatically, and completing of commands args is done by calling \texttt{complete	extunderscore foo()} with arguments \texttt{text}, \texttt{line}, \texttt{begidx}, and \texttt{endidx}. \texttt{text} is the string prefix we are attempting to match: all returned matches must begin with it. \texttt{line} is the current input line with leading whitespace removed, \texttt{begidx} and \texttt{endidx} are the beginning and ending indexes of the prefix text, which could be used to provide different completion depending upon which position the argument is in.

All subclasses of \texttt{Cmd} inherit a predefined \texttt{do	extunderscore help()}. This method, called with an argument \texttt{''bar''}, invokes the corresponding method \texttt{help	extunderscore bar()}, and if that is not present, prints the docstring of \texttt{do	extunderscore bar()}, if available. With no argument, \texttt{do	extunderscore help()} lists all available help topics (that is, all commands with corresponding \texttt{help	extunderscore *()} methods or commands that have docstrings), and also lists any undocumented commands.

\texttt{Cmd. onecmd(str)}

Interpret the argument as though it had been typed in response to the prompt. This may be overridden, but should not normally need to be; see the \texttt{precmd()} and \texttt{postcmd()} methods for useful execution hooks. The return value is a flag indicating whether interpretation of commands by the interpreter should stop. If there is a \texttt{do	extunderscore *()} method for the command \texttt{str}, the return value of that method is returned, otherwise the return value from the \texttt{default()} method is returned.

\texttt{Cmd. emptyline()}

Method called when an empty line is entered in response to the prompt. If this method is not overridden, it repeats the last nonempty command entered.

\texttt{Cmd. default(line)}

Method called on an input line when the command prefix is not
recognized. If this method is not overridden, it prints an error message and returns.

**Cmd.completedefault**(text, line, begidx, endidx)

Method called to complete an input line when no command-specific complete_*() method is available. By default, it returns an empty list.

**Cmd.prcmd**(line)

Hook method executed just before the command line line is interpreted, but after the input prompt is generated and issued. This method is a stub in cmd; it exists to be overridden by subclasses. The return value is used as the command which will be executed by the onecmd() method; the prcmd() implementation may re-write the command or simply return line unchanged.

**Cmd.postcmd**(stop, line)

Hook method executed just after a command dispatch is finished. This method is a stub in cmd; it exists to be overridden by subclasses. line is the command line which was executed, and stop is a flag which indicates whether execution will be terminated after the call to postcmd(); this will be the return value of the onecmd() method. The return value of this method will be used as the new value for the internal flag which corresponds to stop; returning false will cause interpretation to continue.

**Cmd.preloop()

Hook method executed once when cmdloop() is called. This method is a stub in cmd; it exists to be overridden by subclasses.

**Cmd.postloop()

Hook method executed once when cmdloop() is about to return. This method is a stub in cmd; it exists to be overridden by
Instances of subclass Cmd subclasses have some public instance variables:

- **Cmd.prompt**
  - The prompt issued to solicit input.

- **Cmd.identchars**
  - The string of characters accepted for the command prefix.

- **Cmd.lastcmd**
  - The last nonempty command prefix seen.

- **Cmd.cmdqueue**
  - A list of queued input lines. The cmdqueue list is checked in cmdloop() when new input is needed; if it is nonempty, its elements will be processed in order, as if entered at the prompt.

- **Cmd.intro**
  - A string to issue as an intro or banner. May be overridden by giving the cmdloop() method an argument.

- **Cmd.doc_header**
  - The header to issue if the help output has a section for documented commands.

- **Cmd.misc_header**
  - The header to issue if the help output has a section for miscellaneous help topics (that is, there are help_*( ) methods without corresponding do_*( ) methods).

- **Cmd.undoc_header**
  - The header to issue if the help output has a section for undocumented commands (that is, there are do_*( ) methods without corresponding help_*( ) methods).
Cmd. **ruler**

The character used to draw separator lines under the help-message headers. If empty, no ruler line is drawn. It defaults to `'='`.

Cmd. **use_rawinput**

A flag, defaulting to true. If true, `cmdloop()` uses `raw_input()` to display a prompt and read the next command; if false, `sys.stdout.write()` and `sys.stdin.readline()` are used. (This means that by importing `readline`, on systems that support it, the interpreter will automatically support **Emacs**-like line editing and command-history keystrokes.)
23.2. shlex — Simple lexical analysis

New in version 1.5.2.

Source code: Lib/shlex.py

The shlex class makes it easy to write lexical analyzers for simple syntaxes resembling that of the Unix shell. This will often be useful for writing minilanguages, (for example, in run control files for Python applications) or for parsing quoted strings.

Prior to Python 2.7.3, this module did not support Unicode input.

The shlex module defines the following functions:

shlex.split(s[, comments[, posix]])

Split the string s using shell-like syntax. If comments is False (the default), the parsing of comments in the given string will be disabled (setting the commenters attribute of the shlex instance to the empty string). This function operates in POSIX mode by default, but uses non-POSIX mode if the posix argument is false.

New in version 2.3.

Changed in version 2.6: Added the posix parameter.

Note: Since the split() function instantiates a shlex instance, passing None for s will read the string to split from standard input.

The shlex module defines the following class:
A `shlex` instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with `read()` and `readline()` methods, or a string (strings are accepted since Python 2.3). If no argument is given, input will be taken from `sys.stdin`. The second optional argument is a filename string, which sets the initial value of the `infile` attribute. If the `instream` argument is omitted or equal to `sys.stdin`, this second argument defaults to “stdin”. The `posix` argument was introduced in Python 2.3, and defines the operational mode. When `posix` is not true (default), the `shlex` instance will operate in compatibility mode. When operating in POSIX mode, `shlex` will try to be as close as possible to the POSIX shell parsing rules.

**See also:**

**Module ConfigParser**

Parser for configuration files similar to the Windows `.ini` files.
23.2.1. shlex Objects

A shlex instance has the following methods:

shlex.get_token()

Return a token. If tokens have been stacked using push_token(), pop a token off the stack. Otherwise, read one from the input stream. If reading encounters an immediate end-of-file, eof is returned (the empty string ("") in non-POSIX mode, and None in POSIX mode).

shlex.push_token(str)

Push the argument onto the token stack.

shlex.read_token()

Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

shlex.sourcehook(filename)

When shlex detects a source request (see source below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (such as sys.stdin), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor
handles `#include "file.h"`).

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with `open()` called on it to yield the second component. (Note: this is the reverse of the order of arguments in instance initialization!)

This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding ‘close’ hook, but a `shlex` instance will call the `close()` method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the `push_source()` and `pop_source()` methods.

```
shlex.push_source(stream[, filename])
```

Push an input source stream onto the input stack. If the filename argument is specified it will later be available for use in error messages. This is the same method used internally by the `sourcehook()` method.

*New in version 2.1.*

```
shlex.pop_source()
```

Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer reaches EOF on a stacked input stream.

*New in version 2.1.*

```
shlex.error_leader([file[, line]])
```

This method generates an error message leader in the format of a Unix C compiler error label; the format is `'"%s", line %d: '`,
where the %s is replaced with the name of the current source file and the %d with the current input line number (the optional arguments can be used to override these).

This convenience is provided to encourage shlex users to generate error messages in the standard, parseable format understood by Emacs and other Unix tools.

Instances of shlex subclasses have some public instance variables which either control lexical analysis or can be used for debugging:

shlex.commenters
The string of characters that are recognized as comment beginners. All characters from the comment beginner to end of line are ignored. Includes just '#' by default.

shlex.wordchars
The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumerics and underscore.

shlex.whitespace
Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

shlex.escape
Characters that will be considered as escape. This will be only used in POSIX mode, and includes just '\' by default.

New in version 2.3.

shlex.quotes
Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By
default, includes ASCII single and double quotes.

**shlex.escapedquotes**

Characters in *quotes* that will interpret escape characters defined in *escape*. This is only used in POSIX mode, and includes just `'` by default.

*New in version 2.3.*

**shlex.whitespace_split**

If *True*, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with *shlex*, getting tokens in a similar way to shell arguments.

*New in version 2.3.*

**shlex.infile**

The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

**shlex.instream**

The input stream from which this *shlex* instance is reading characters.

**shlex.source**

This attribute is *None* by default. If you assign a string to it, that string will be recognized as a lexical-level inclusion request similar to the *source* keyword in various shells. That is, the immediately following token will be opened as a filename and input will be taken from that stream until EOF, at which point the *close()* method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

**shlex.debug**
If this attribute is numeric and 1 or more, a `shlex` instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.

`shlex.lineno`
Source line number (count of newlines seen so far plus one).

`shlex.token`
The token buffer. It may be useful to examine this when catching exceptions.

`shlex.eof`
Token used to determine end of file. This will be set to the empty string (''), in non-POSIX mode, and to `None` in POSIX mode.

*New in version 2.3.*
23.2.2. Parsing Rules

When operating in non-POSIX mode, `shlex` will try to obey to the following rules.

- Quote characters are not recognized within words (`Do"Not"Separate` is parsed as the single word `Do"Not"Separate`);
- Escape characters are not recognized;
- Enclosing characters in quotes preserve the literal value of all characters within the quotes;
- Closing quotes separate words (`"Do"Separate` is parsed as `Do` and `Separate`);
- If `whitespace_split` is `False`, any character not declared to be a word character, whitespace, or a quote will be returned as a single-character token. If it is `True`, `shlex` will only split words in whitespaces;
- EOF is signaled with an empty string (`''`);
- It’s not possible to parse empty strings, even if quoted.

When operating in POSIX mode, `shlex` will try to obey to the following parsing rules.

- Quotes are stripped out, and do not separate words (`"Do"Not"Separate` is parsed as the single word `DoNotSeparate`);
- Non-quoted escape characters (e.g. `'\'`) preserve the literal value of the next character that follows;
- Enclosing characters in quotes which are not part of `escapedquotes` (e.g. `'''`) preserve the literal value of all characters within the quotes;
- Enclosing characters in quotes which are part of `escapedquotes` (e.g. `'''`) preserves the literal value of all characters within the quotes, with the exception of the characters mentioned in
The escape characters retain its special meaning only when followed by the quote in use, or the escape character itself. Otherwise the escape character will be considered a normal character.

- EOF is signaled with a `None` value;
- Quoted empty strings (`''`) are allowed;
24. Graphical User Interfaces with Tk

Tk/Tcl has long been an integral part of Python. It provides a robust and platform independent windowing toolkit, that is available to Python programmers using the Tkinter module, and its extensions, the Tix and the ttk modules.

The Tkinter module is a thin object-oriented layer on top of Tcl/Tk. To use Tkinter, you don’t need to write Tcl code, but you will need to consult the Tk documentation, and occasionally the Tcl documentation. Tkinter is a set of wrappers that implement the Tk widgets as Python classes. In addition, the internal module _tkinter provides a threadsafe mechanism which allows Python and Tcl to interact.

Tkinter’s chief virtues are that it is fast, and that it usually comes bundled with Python. Although its standard documentation is weak, good material is available, which includes: references, tutorials, a book and others. Tkinter is also famous for having an outdated look and feel, which has been vastly improved in Tk 8.5. Nevertheless, there are many other GUI libraries that you could be interested in. For more information about alternatives, see the Other Graphical User Interface Packages section.

- 24.1. Tkinter — Python interface to Tcl/Tk
  - 24.1.1. Tkinter Modules
  - 24.1.2. Tkinter Life Preserver
    - 24.1.2.1. How To Use This Section
    - 24.1.2.2. A Simple Hello World Program
  - 24.1.3. A (Very) Quick Look at Tcl/Tk
  - 24.1.4. Mapping Basic Tk into Tkinter
• 24.1.5. How Tk and Tkinter are Related
• 24.1.6. Handy Reference
  ■ 24.1.6.1. Setting Options
  ■ 24.1.6.2. The Packer
  ■ 24.1.6.3. Packer Options
  ■ 24.1.6.4. Coupling Widget Variables
  ■ 24.1.6.5. The Window Manager
  ■ 24.1.6.6. Tk Option Data Types
  ■ 24.1.6.7. Bindings and Events
  ■ 24.1.6.8. The index Parameter
  ■ 24.1.6.9. Images
• 24.1.7. File Handlers
• 24.2. **ttk** — Tk themed widgets
  • 24.2.1. Using Ttk
  • 24.2.2. Ttk Widgets
  • 24.2.3. Widget
    ■ 24.2.3.1. Standard Options
    ■ 24.2.3.2. Scrollable Widget Options
    ■ 24.2.3.3. Label Options
    ■ 24.2.3.4. Compatibility Options
    ■ 24.2.3.5. Widget States
    ■ 24.2.3.6. ttk.Widget
  • 24.2.4. Combobox
    ■ 24.2.4.1. Options
    ■ 24.2.4.2. Virtual events
    ■ 24.2.4.3. ttk.Combobox
  • 24.2.5. Notebook
    ■ 24.2.5.1. Options
    ■ 24.2.5.2. Tab Options
    ■ 24.2.5.3. Tab Identifiers
    ■ 24.2.5.4. Virtual Events
    ■ 24.2.5.5. ttk.Notebook
  • 24.2.6. Progressbar
    ■ 24.2.6.1. Options
- 24.2.6.2. ttk.Progressbar
- 24.2.7. Separator
  - 24.2.7.1. Options
- 24.2.8. Sizegrip
  - 24.2.8.1. Platform-specific notes
  - 24.2.8.2. Bugs
- 24.2.9. Treeview
  - 24.2.9.1. Options
  - 24.2.9.2. Item Options
  - 24.2.9.3. Tag Options
  - 24.2.9.4. Column Identifiers
  - 24.2.9.5. Virtual Events
  - 24.2.9.6. ttk.Treeview
- 24.2.10. Ttk Styling
  - 24.2.10.1. Layouts
- **24.3. Tix — Extension widgets for Tk**
  - 24.3.1. Using Tix
  - 24.3.2. Tix Widgets
    - 24.3.2.1. Basic Widgets
    - 24.3.2.2. File Selectors
    - 24.3.2.3. Hierarchical ListBox
    - 24.3.2.4. Tabular ListBox
    - 24.3.2.5. Manager Widgets
    - 24.3.2.6. Image Types
    - 24.3.2.7. Miscellaneous Widgets
    - 24.3.2.8. Form Geometry Manager
  - 24.3.3. Tix Commands
- **24.4. ScrolledText — Scrolled Text Widget**
- **24.5. turtle — Turtle graphics for Tk**
  - 24.5.1. Introduction
  - 24.5.2. Overview over available Turtle and Screen methods
    - 24.5.2.1. Turtle methods
    - 24.5.2.2. Methods of TurtleScreen/Screen
  - 24.5.3. Methods of RawTurtle/Turtle and corresponding
functions

- 24.5.3.1. Turtle motion
- 24.5.3.2. Tell Turtle's state
- 24.5.3.3. Settings for measurement
- 24.5.3.4. Pen control
  - 24.5.3.4.1. Drawing state
  - 24.5.3.4.2. Color control
  - 24.5.3.4.3. Filling
  - 24.5.3.4.4. More drawing control
- 24.5.3.5. Turtle state
  - 24.5.3.5.1. Visibility
  - 24.5.3.5.2. Appearance
- 24.5.3.6. Using events
- 24.5.3.7. Special Turtle methods
- 24.5.3.8. Excursus about the use of compound shapes

- 24.5.4. Methods of TurtleScreen/Screen and corresponding functions
  - 24.5.4.1. Window control
  - 24.5.4.2. Animation control
  - 24.5.4.3. Using screen events
  - 24.5.4.4. Settings and special methods
  - 24.5.4.5. Methods specific to Screen, not inherited from TurtleScreen

- 24.5.5. The public classes of the module turtle

- 24.5.6. Help and configuration
  - 24.5.6.1. How to use help
  - 24.5.6.2. Translation of docstrings into different languages
  - 24.5.6.3. How to configure Screen and Turtles

- 24.5.7. Demo scripts

- 24.6. IDLE
  - 24.6.1. Menus
    - 24.6.1.1. File menu (Shell and Editor)
    - 24.6.1.2. Edit menu (Shell and Editor)
- 24.6.1.3. Format menu (Editor window only)
- 24.6.1.4. Run menu (Editor window only)
- 24.6.1.5. Shell menu (Shell window only)
- 24.6.1.6. Debug menu (Shell window only)
- 24.6.1.7. Options menu (Shell and Editor)
- 24.6.1.8. Window menu (Shell and Editor)
- 24.6.1.9. Help menu (Shell and Editor)
- 24.6.1.10. Context Menus

- 24.6.2. Editing and navigation
  - 24.6.2.1. Automatic indentation
  - 24.6.2.2. Completions
  - 24.6.2.3. Calltips
  - 24.6.2.4. Python Shell window
  - 24.6.2.5. Text colors

- 24.6.3. Startup and code execution
  - 24.6.3.1. Command line usage
  - 24.6.3.2. IDLE-console differences
  - 24.6.3.3. Running without a subprocess

- 24.6.4. Help and preferences
  - 24.6.4.1. Additional help sources
  - 24.6.4.2. Setting preferences
  - 24.6.4.3. Extensions

- 24.7. Other Graphical User Interface Packages
The Tkinter module ("Tk interface") is the standard Python interface to the Tk GUI toolkit. Both Tk and Tkinter are available on most Unix platforms, as well as on Windows systems. (Tk itself is not part of Python; it is maintained at ActiveState.)

Note: Tkinter has been renamed to tkinter in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

See also:

Python Tkinter Resources
The Python Tkinter Topic Guide provides a great deal of information on using Tk from Python and links to other sources of information on Tk.

TKDocs
Extensive tutorial plus friendlier widget pages for some of the widgets.

Tkinter reference: a GUI for Python
On-line reference material.

Tkinter docs from effbot
Online reference for tkinter supported by effbot.org.

Tcl/Tk manual
Official manual for the latest tcl/tk version.

Programming Python
Book by Mark Lutz, has excellent coverage of Tkinter.

Modern Tkinter for Busy Python Developers
Book by Mark Rozerman about building attractive and modern graphical user interfaces with Python and Tkinter.

**Python and Tkinter Programming**

24.1.1. Tkinter Modules

Most of the time, the Tkinter module is all you really need, but a number of additional modules are available as well. The Tk interface is located in a binary module named _tkinter. This module contains the low-level interface to Tk, and should never be used directly by application programmers. It is usually a shared library (or DLL), but might in some cases be statically linked with the Python interpreter.

In addition to the Tk interface module, Tkinter includes a number of Python modules. The two most important modules are the Tkinter module itself, and a module called Tkconstants. The former automatically imports the latter, so to use Tkinter, all you need to do is to import one module:

```python
import Tkinter
```

Or, more often:

```python
from Tkinter import *
```

```python
class Tkinter.Tk(screenName=None, baseName=None, className='Tk', useTk=1)

    The Tk class is instantiated without arguments. This creates a toplevel widget of Tk which usually is the main window of an application. Each instance has its own associated Tcl interpreter.

    Changed in version 2.4: The useTk parameter was added.
```

```python
Tkinter.Tcl(screenName=None, baseName=None, className='Tk', useTk=0)

    The Tcl() function is a factory function which creates an object much like that created by the Tk class, except that it does not
```
initialize the Tk subsystem. This is most often useful when driving the Tcl interpreter in an environment where one doesn’t want to create extraneous toplevel windows, or where one cannot (such as Unix/Linux systems without an X server). An object created by the Tcl() object can have a Toplevel window created (and the Tk subsystem initialized) by calling its loadtk() method.

New in version 2.4.

Other modules that provide Tk support include:

**ScrolledText**
Text widget with a vertical scroll bar built in.

**tkColorChooser**
Dialog to let the user choose a color.

**tkCommonDialog**
Base class for the dialogs defined in the other modules listed here.

**tkFileDialog**
Common dialogs to allow the user to specify a file to open or save.

**tkFont**
Utilities to help work with fonts.

**tkMessageBox**
Access to standard Tk dialog boxes.

**tkSimpleDialog**
Basic dialogs and convenience functions.

**TkDnd**
Drag-and-drop support for Tkinter. This is experimental and should become deprecated when it is replaced with the Tk DND.

**turtle**
Turtle graphics in a Tk window.
These have been renamed as well in Python 3; they were all made submodules of the new `tkinter` package.
24.1.2. Tkinter Life Preserver

This section is not designed to be an exhaustive tutorial on either Tk or Tkinter. Rather, it is intended as a stop gap, providing some introductory orientation on the system.

Credits:

- Tkinter was written by Steen Lumholt and Guido van Rossum.
- Tk was written by John Ousterhout while at Berkeley.
- This Life Preserver was written by Matt Conway at the University of Virginia.
- The html rendering, and some liberal editing, was produced from a FrameMaker version by Ken Manheimer.
- Fredrik Lundh elaborated and revised the class interface descriptions, to get them current with Tk 4.2.
- Mike Clarkson converted the documentation to LaTeX, and compiled the User Interface chapter of the reference manual.

24.1.2.1. How To Use This Section

This section is designed in two parts: the first half (roughly) covers background material, while the second half can be taken to the keyboard as a handy reference.

When trying to answer questions of the form “how do I do blah”, it is often best to find out how to do”blah” in straight Tk, and then convert this back into the corresponding Tkinter call. Python programmers can often guess at the correct Python command by looking at the Tk documentation. This means that in order to use Tkinter, you will have to know a little bit about Tk. This document can’t fulfill that role, so the best we can do is point you to the best documentation that exists. Here are some hints:
The authors strongly suggest getting a copy of the Tk man pages. Specifically, the man pages in the `man` directory are most useful. The `man3` man pages describe the C interface to the Tk library and thus are not especially helpful for script writers.

Addison-Wesley publishes a book called Tcl and the Tk Toolkit by John Ousterhout (ISBN 0-201-63337-X) which is a good introduction to Tcl and Tk for the novice. The book is not exhaustive, and for many details it defers to the man pages.

Tkinter.py is a last resort for most, but can be a good place to go when nothing else makes sense.

See also:

ActiveState Tcl Home Page
The Tk/Tcl development is largely taking place at ActiveState.

Tcl and the Tk Toolkit
The book by John Ousterhout, the inventor of Tcl.

Practical Programming in Tcl and Tk
Brent Welch’s encyclopedic book.

24.1.2.2. A Simple Hello World Program

```python
from Tkinter import *

class Application(Frame):
    def say_hi(self):
        print "hi there, everyone!"

    def createWidgets(self):
        self.QUIT = Button(self)
        self.QUIT["text"] = "QUIT"
        self.QUIT["fg"] = "red"
        self.QUIT["command"] = self.quit

        self.QUIT.pack({"side": "left"})
```
self.hi_there = Button(self)
self.hi_there["text"] = "Hello",
self.hi_there["command"] = self.say_hi

self.hi_there.pack({"side": "left"})

def __init__(self, master=None):
    Frame.__init__(self, master)
    self.pack()
    self.createWidgets()

root = Tk()
app = Application(master=root)
app.mainloop()
root.destroy()
24.1.3. A (Very) Quick Look at Tcl/Tk

The class hierarchy looks complicated, but in actual practice, application programmers almost always refer to the classes at the very bottom of the hierarchy.

Notes:

- These classes are provided for the purposes of organizing certain functions under one namespace. They aren’t meant to be instantiated independently.
- The Tk class is meant to be instantiated only once in an application. Application programmers need not instantiate one explicitly, the system creates one whenever any of the other classes are instantiated.
- The Widget class is not meant to be instantiated, it is meant only for subclassing to make “real” widgets (in C++, this is called an ‘abstract class’).

To make use of this reference material, there will be times when you will need to know how to read short passages of Tk and how to identify the various parts of a Tk command. (See section Mapping Basic Tk into Tkinter for the Tkinter equivalents of what’s below.)

Tk scripts are Tcl programs. Like all Tcl programs, Tk scripts are just lists of tokens separated by spaces. A Tk widget is just its class, the options that help configure it, and the actions that make it do useful things.

To make a widget in Tk, the command is always of the form:

```
classCommand newPathname options
```

`classCommand`
denotes which kind of widget to make (a button, a label, a menu...)

newPathname

is the new name for this widget. All names in Tk must be unique. To help enforce this, widgets in Tk are named with pathnames, just like files in a file system. The top level widget, the root, is called . (period) and children are delimited by more periods. For example, MyApp.controlPanel.okButton might be the name of a widget.

options

configure the widget’s appearance and in some cases, its behavior. The options come in the form of a list of flags and values. Flags are preceded by a ‘-’, like Unix shell command flags, and values are put in quotes if they are more than one word.

For example:

```
button .fred -fg red -text "hi there"
```

Once created, the pathname to the widget becomes a new command. This new widget command is the programmer’s handle for getting the new widget to perform some action. In C, you’d express this as someAction(fred, someOptions), in C++, you would express this as fred.someAction(someOptions), and in Tk, you say:

```
.fred someAction someOptions
```

Note that the object name, .fred, starts with a dot.

As you’d expect, the legal values for someAction will depend on the
widget's class: .fred disable works if fred is a button (fred gets greyed out), but does not work if fred is a label (disabling of labels is not supported in Tk).

The legal values of someOptions is action dependent. Some actions, like disable, require no arguments, others, like a text-entry box's delete command, would need arguments to specify what range of text to delete.
24.1.4. Mapping Basic Tk into Tkinter

Class commands in Tk correspond to class constructors in Tkinter.

```
button .fred
====> fred = Button()
```

The master of an object is implicit in the new name given to it at creation time. In Tkinter, masters are specified explicitly.

```
button .panel.fred
====> fred = Button(panel)
```

The configuration options in Tk are given in lists of hyphenated tags followed by values. In Tkinter, options are specified as keyword-arguments in the instance constructor, and keyword-args for configure calls or as instance indices, in dictionary style, for established instances. See section Setting Options on setting options.

```
button .fred -fg red
.fred configure -fg red
====> fred = Button(panel, fg = "red"
====> fred["fg"] = red
0R ==> fred.config(fg = "red")
```

In Tk, to perform an action on a widget, use the widget name as a command, and follow it with an action name, possibly with arguments (options). In Tkinter, you call methods on the class instance to invoke actions on the widget. The actions (methods) that a given widget can perform are listed in the Tkinter.py module.

```
.fred invoke
====> fred.invoke()
```

To give a widget to the packer (geometry manager), you call pack with optional arguments. In Tkinter, the Pack class holds all this functionality, and the various forms of the pack command are implemented as methods. All widgets in Tkinter are subclassed.
from the Packer, and so inherit all the packing methods. See the Tix module documentation for additional information on the Form geometry manager.

```
pack .fred -side left       =====>  fred.pack(side = "left")
```
24.1.5. How Tk and Tkinter are Related

From the top down:

Your App Here (Python)
   A Python application makes a Tkinter call.

Tkinter (Python Module)
   This call (say, for example, creating a button widget), is implemented in the Tkinter module, which is written in Python. This Python function will parse the commands and the arguments and convert them into a form that makes them look as if they had come from a Tk script instead of a Python script.

tkinter (C)
   These commands and their arguments will be passed to a C function in the tkinter - note the lowercase - extension module.

Tk Widgets (C and Tcl)
   This C function is able to make calls into other C modules, including the C functions that make up the Tk library. Tk is implemented in C and some Tcl. The Tcl part of the Tk widgets is used to bind certain default behaviors to widgets, and is executed once at the point where the Python Tkinter module is imported. (The user never sees this stage).

Tk (C)
   The Tk part of the Tk Widgets implement the final mapping to ...

Xlib (C)
   the Xlib library to draw graphics on the screen.
24.1.6. Handy Reference

24.1.6.1. Setting Options

Options control things like the color and border width of a widget. Options can be set in three ways:

At object creation time, using keyword arguments

```python
fred = Button(self, fg = "red", bg = "blue")
```

After object creation, treating the option name like a dictionary index

```python
fred["fg"] = "red"
fred["bg"] = "blue"
```

Use the config() method to update multiple attrs subsequent to object creation

```python
fred.config(fg = "red", bg = "blue")
```

For a complete explanation of a given option and its behavior, see the Tk man pages for the widget in question.

Note that the man pages list “STANDARD OPTIONS” and “WIDGET SPECIFIC OPTIONS” for each widget. The former is a list of options that are common to many widgets, the latter are the options that are idiosyncratic to that particular widget. The Standard Options are documented on the `options(3)` man page.

No distinction between standard and widget-specific options is made in this document. Some options don’t apply to some kinds of widgets. Whether a given widget responds to a particular option depends on the class of the widget; buttons have a `command` option, labels do not.
The options supported by a given widget are listed in that widget’s man page, or can be queried at runtime by calling the `config()` method without arguments, or by calling the `keys()` method on that widget. The return value of these calls is a dictionary whose key is the name of the option as a string (for example, `'relief'`) and whose values are 5-tuples.

Some options, like `bg` are synonyms for common options with long names (`bg` is shorthand for “background”). Passing the `config()` method the name of a shorthand option will return a 2-tuple, not 5-tuple. The 2-tuple passed back will contain the name of the synonym and the “real” option (such as `('bg', 'background')`).

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>option name</td>
<td><code>'relief'</code></td>
</tr>
<tr>
<td>1</td>
<td>option name for database lookup</td>
<td><code>'relief'</code></td>
</tr>
<tr>
<td>2</td>
<td>option class for database lookup</td>
<td><code>'Relief'</code></td>
</tr>
<tr>
<td>3</td>
<td>default value</td>
<td><code>'raised'</code></td>
</tr>
<tr>
<td>4</td>
<td>current value</td>
<td><code>'groove'</code></td>
</tr>
</tbody>
</table>

Example:

```python
>>> print fred.config()
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

Of course, the dictionary printed will include all the options available and their values. This is meant only as an example.

### 24.1.6.2. The Packer

The packer is one of Tk’s geometry-management mechanisms. Geometry managers are used to specify the relative positioning of the positioning of widgets within their container - their mutual `master`. In contrast to the more cumbersome `placer` (which is used less
commonly, and we do not cover here), the packer takes qualitative relationship specification - *above, to the left of, filling*, etc - and works everything out to determine the exact placement coordinates for you.

The size of any *master* widget is determined by the size of the “slave widgets” inside. The packer is used to control where slave widgets appear inside the master into which they are packed. You can pack widgets into frames, and frames into other frames, in order to achieve the kind of layout you desire. Additionally, the arrangement is dynamically adjusted to accommodate incremental changes to the configuration, once it is packed.

Note that widgets do not appear until they have had their geometry specified with a geometry manager. It's a common early mistake to leave out the geometry specification, and then be surprised when the widget is created but nothing appears. A widget will appear only after it has had, for example, the packer’s `pack()` method applied to it.

The `pack()` method can be called with keyword-option/value pairs that control where the widget is to appear within its container, and how it is to behave when the main application window is resized. Here are some examples:

```python
fred.pack()  # defaults to side = "top"
fred.pack(side = "left")
fred.pack(expand = 1)
```

### 24.1.6.3. Packer Options

For more extensive information on the packer and the options that it can take, see the man pages and page 183 of John Ousterhout's book.

anchor

Anchor type. Denotes where the packer is to place each slave in
its parcel.

expand

Boolean, 0 or 1.

fill

Legal values: 'x', 'y', 'both', 'none'.

ipadx and ipady

A distance - designating internal padding on each side of the slave widget.

padx and pady

A distance - designating external padding on each side of the slave widget.

side

Legal values are: 'left', 'right', 'top', 'bottom'.

24.1.6.4. Coupling Widget Variables

The current-value setting of some widgets (like text entry widgets) can be connected directly to application variables by using special options. These options are variable, textvariable, onvalue, offvalue, and value. This connection works both ways: if the variable changes for any reason, the widget it’s connected to will be updated to reflect the new value.

Unfortunately, in the current implementation of Tkinter it is not possible to hand over an arbitrary Python variable to a widget through a variable or textvariable option. The only kinds of variables for which this works are variables that are subclassed from a class called Variable, defined in the Tkinter module.

There are many useful subclasses of Variable already defined: StringVar, IntVar, DoubleVar, and BooleanVar. To read the current
value of such a variable, call the `get()` method on it, and to change its value you call the `set()` method. If you follow this protocol, the widget will always track the value of the variable, with no further intervention on your part.

For example:

```python
class App(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()

        self.entrythingy = Entry()
        self.entrythingy.pack()

        # here is the application variable
        self.contents = StringVar()
        # set it to some value
        self.contents.set("this is a variable")
        # tell the entry widget to watch this variable
        self.entrythingy["textvariable"] = self.contents

        # and here we get a callback when the user hits return.
        # we will have the program print out the value of the
        # application variable when the user hits return
        self.entrythingy.bind('<Key-Return>',
            self.print_contents)

    def print_contents(self, event):
        print "hi. contents of entry is now ----->", \
            self.contents.get()
```

### 24.1.6.5. The Window Manager

In Tk, there is a utility command, `wm`, for interacting with the window manager. Options to the `wm` command allow you to control things like titles, placement, icon bitmaps, and the like. In Tkinter, these commands have been implemented as methods on the `Wm` class. Toplevel widgets are subclassed from the `Wm` class, and so can call
the `wm` methods directly.

To get at the toplevel window that contains a given widget, you can often just refer to the widget's master. Of course if the widget has been packed inside of a frame, the master won't represent a toplevel window. To get at the toplevel window that contains an arbitrary widget, you can call the `_root()` method. This method begins with an underscore to denote the fact that this function is part of the implementation, and not an interface to Tk functionality.

Here are some examples of typical usage:

```python
from Tkinter import *
class App(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()

    # create the application
    myapp = App()

    # here are method calls to the window manager class
    myapp.master.title("My Do-Nothing Application")
    myapp.master.maxsize(1000, 400)

    # start the program
    myapp.mainloop()
```

### 24.1.6.6. Tk Option Data Types

**anchor**

Legal values are points of the compass: "n", "ne", "e", "se", "s", "sw", "w", "nw", and also "center".

**bitmap**

There are eight built-in, named bitmaps: 'error', 'gray25', 
To specify an X bitmap filename, give the full path to the file, preceded with an @, as in @/usr/contrib(bitmap/gumby.bit).

**boolean**

You can pass integers 0 or 1 or the strings "yes" or "no".

**callback**

This is any Python function that takes no arguments. For example:

```python
def print_it():
    print "hi there"
fred["command"] = print_it
```

**color**

Colors can be given as the names of X colors in the rgb.txt file, or as strings representing RGB values in 4 bit: "#RGB", 8 bit: "#RRGGBB", 12 bit "#RRRGGGBBB", or 16 bit "#RRRGGGGGBBBB" ranges, where R,G,B here represent any legal hex digit. See page 160 of Ousterhout's book for details.

**cursor**

The standard X cursor names from cursorfont.h can be used, without the xc prefix. For example to get a hand cursor (xc_hand2), use the string "hand2". You can also specify a bitmap and mask file of your own. See page 179 of Ousterhout's book.

**distance**

Screen distances can be specified in either pixels or absolute distances. Pixels are given as numbers and absolute distances as strings, with the trailing character denoting units: c for centimetres, i for inches, m for millimetres, p for printer's points. For example, 3.5 inches is expressed as "3.5i".

**font**
Tk uses a list font name format, such as \{courier 10 bold\}. Font sizes with positive numbers are measured in points; sizes with negative numbers are measured in pixels.

**geometry**
This is a string of the form \texttt{width\times height}, where width and height are measured in pixels for most widgets (in characters for widgets displaying text). For example: \texttt{fred["geometry"] = "200\times100"}.

**justify**
Legal values are the strings: \texttt{"left"}, \texttt{"center"}, \texttt{"right"}, and \texttt{"fill"}.

**region**
This is a string with four space-delimited elements, each of which is a legal distance (see above). For example: \texttt{"2 3 4 5"} and \texttt{"3i 2i 4.5i 2i"} and \texttt{"3c 2c 4c 10.43c"} are all legal regions.

**relief**
Determines what the border style of a widget will be. Legal values are: \texttt{"raised"}, \texttt{"sunken"}, \texttt{"flat"}, \texttt{"groove"}, and \texttt{"ridge"}.

**scrollcommand**
This is almost always the \texttt{set()} method of some scrollbar widget, but can be any widget method that takes a single argument. Refer to the file \texttt{Demo/tkinter/matt/canvas-with-scrollbars.py} in the Python source distribution for an example.

**wrap**
Must be one of: \texttt{"none"}, \texttt{"char"}, or \texttt{"word"}.

### 24.1.6.7. Bindings and Events

The bind method from the widget command allows you to watch for certain events and to have a callback function trigger when that
event type occurs. The form of the bind method is:

```
def bind(self, sequence, func, add=' '):
```

where:

sequence

is a string that denotes the target kind of event. (See the bind man page and page 201 of John Ousterhout’s book for details).

func

is a Python function, taking one argument, to be invoked when the event occurs. An Event instance will be passed as the argument. (Functions deployed this way are commonly known as callbacks.)

add

is optional, either ' ' or '+'. Passing an empty string denotes that this binding is to replace any other bindings that this event is associated with. Passing a '+' means that this function is to be added to the list of functions bound to this event type.

For example:

```
def turnRed(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turnRed)
```

Notice how the widget field of the event is being accessed in the turnRed() callback. This field contains the widget that caught the X event. The following table lists the other event fields you can access, and how they are denoted in Tk, which can be useful when referring to the Tk man pages.

```
<table>
<thead>
<tr>
<th>Tk</th>
<th>Tkinter Event Field</th>
<th>Tk</th>
<th>Tkinter Event Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>%f</td>
<td>focus</td>
<td>%A</td>
<td>char</td>
</tr>
</tbody>
</table>
```
24.1.6.8. The index Parameter

A number of widgets require "index" parameters to be passed. These are used to point at a specific place in a Text widget, or to particular characters in an Entry widget, or to particular menu items in a Menu widget.

Entry widget indexes (index, view index, etc.)

Entry widgets have options that refer to character positions in the text being displayed. You can use these `Tkinter` functions to access these special points in text widgets:

- `AtEnd()`
  refers to the last position in the text

- `AtInsert()`
  refers to the point where the text cursor is

- `AtSelFirst()`
  indicates the beginning point of the selected text

- `AtSelLast()`
  denotes the last point of the selected text and finally

- `At(x[, y])`
  refers to the character at pixel location x, y (with y not used in the case of a text entry widget, which contains a single line of text).

Text widget indexes
The index notation for Text widgets is very rich and is best described in the Tk man pages.

Menu indexes (menu.invoke(), menu.entryconfig(), etc.)
Some options and methods for menus manipulate specific menu entries. Anytime a menu index is needed for an option or a parameter, you may pass in:

- an integer which refers to the numeric position of the entry in the widget, counted from the top, starting with 0;
- the string "active", which refers to the menu position that is currently under the cursor;
- the string "last" which refers to the last menu item;
- An integer preceded by @, as in @6, where the integer is interpreted as a y pixel coordinate in the menu's coordinate system;
- the string "none", which indicates no menu entry at all, most often used with menu.activate() to deactivate all entries, and finally,
- a text string that is pattern matched against the label of the menu entry, as scanned from the top of the menu to the bottom. Note that this index type is considered after all the others, which means that matches for menu items labelled last, active, or none may be interpreted as the above literals, instead.

### 24.1.6.9. Images

Bitmap/Pixelmap images can be created through the subclasses of Tkinter.Image:

- **BitmapImage** can be used for X11 bitmap data.
- **PhotoImage** can be used for GIF and PPM/PGM color bitmaps.
Either type of image is created through either the file or the data option (other options are available as well).

The image object can then be used wherever an image option is supported by some widget (e.g. labels, buttons, menus). In these cases, Tk will not keep a reference to the image. When the last Python reference to the image object is deleted, the image data is deleted as well, and Tk will display an empty box wherever the image was used.
24.1.7. File Handlers

Tk allows you to register and unregister a callback function which will be called from the Tk mainloop when I/O is possible on a file descriptor. Only one handler may be registered per file descriptor. Example code:

```python
import Tkinter
widget = Tkinter.Tk()
mask = Tkinter.READABLE | Tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

This feature is not available on Windows.

Since you don’t know how many bytes are available for reading, you may not want to use the `BufferedIOBase` or `TextIOBase` `read()` or `readline()` methods, since these will insist on reading a predefined number of bytes. For sockets, the `recv()` or `recvfrom()` methods will work fine; for other files, use raw reads or `os.read(file.fileno(), maxbytecount)`.  

`Widget.tk.createfilehandler(file, mask, func)`

Registers the file handler callback function `func`. The `file` argument may either be an object with a `fileno()` method (such as a file or socket object), or an integer file descriptor. The `mask` argument is an ORed combination of any of the three constants below. The callback is called as follows:

```python
callback(file, mask)
```

`Widget.tk.deletefilehandler(file)`

Unregisters a file handler.
Constants used in the *mask* arguments.
24. Graphical User Interfaces with Tk
24.2. `ttk` — Tk themed widgets

The `ttk` module provides access to the Tk themed widget set, which has been introduced in Tk 8.5. If Python is not compiled against Tk 8.5 code may still use this module as long as Tile is installed. However, some features provided by the new Tk, like anti-aliased font rendering under X11, window transparency (on X11 you will need a composition window manager) will be missing.

The basic idea of `ttk` is to separate, to the extent possible, the code implementing a widget's behavior from the code implementing its appearance.

See also:

Tk Widget Styling Support
The document which brought up theming support for Tk
24.2.1. Using Ttk

To start using Ttk, import its module:

```python
import ttk
```

But code like this:

```python
from Tkinter import *
```

may optionally want to use this:

```python
from Tkinter import *
from ttk import *
```

And then several `ttk` widgets (Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale and Scrollbar) will automatically substitute for the Tk widgets.

This has the direct benefit of using the new widgets, giving better look & feel across platforms, but be aware that they are not totally compatible. The main difference is that widget options such as “fg”, “bg” and others related to widget styling are no longer present in Ttk widgets. Use `ttk.Style` to achieve the same (or better) styling.

**See also:**

**Converting existing applications to use the Tile widgets**

A text which talks in Tcl terms about differences typically found when converting applications to use the new widgets.
24.2.2. Ttk Widgets

Ttk comes with 17 widgets, 11 of which already exist in Tkinter: Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale and Scrollbar. The 6 new widget classes are: Combobox, Notebook, Progressbar, Separator, Sizegrip and Treeview. All of these classes are subclasses of Widget.

As said previously, you will notice changes in look-and-feel as well in the styling code. To demonstrate the latter, a very simple example is shown below.

Tk code:

```python
l1 = Tkinter.Label(text="Test", fg="black", bg="white")
l2 = Tkinter.Label(text="Test", fg="black", bg="white")
```

Corresponding Ttk code:

```python
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")
l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

For more information about TtkStyling read the style class documentation.
24.2.3. Widget

`ttk.Widget` defines standard options and methods supported by Tk themed widgets and is not supposed to be directly instantiated.

24.2.3.1. Standard Options

All the `ttk` widgets accept the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Specifies the window class. The class is used when querying the option database for the window's other options, to determine the default bindtags for the window, and to select the widget's default layout and style. This is a read-only option which may only be specified when the window is created.</td>
</tr>
<tr>
<td>cursor</td>
<td>Specifies the mouse cursor to be used for the widget. If set to the empty string (the default), the cursor is inherited from the parent widget.</td>
</tr>
<tr>
<td>takefocus</td>
<td>Determines whether the window accepts the focus during keyboard traversal. 0, 1 or an empty string is returned. If 0, the window should be skipped entirely during keyboard traversal. If 1, the window should receive the input focus as long as it is viewable. An empty string means that the traversal scripts make the decision about whether or not to focus on the window.</td>
</tr>
<tr>
<td>style</td>
<td>May be used to specify a custom widget style.</td>
</tr>
</tbody>
</table>

24.2.3.2. Scrollable Widget Options

The following options are supported by widgets that are controlled by a scrollbar.
### xscrollcommand

Used to communicate with horizontal scrollbars.

When the view in the widget's window changes, the widget will generate a Tcl command based on the scrollcommand.

Usually this option consists of the `Scrollbar.set()` method of some scrollbar. This will cause the scrollbar to be updated whenever the view in the window changes.

### yscrollcommand

Used to communicate with vertical scrollbars. For more information, see above.

### 24.2.3.3. Label Options

The following options are supported by labels, buttons and other button-like widgets.

<table>
<thead>
<tr>
<th>option</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>Specifies a text string to be displayed inside the widget.</td>
</tr>
<tr>
<td>textvariable</td>
<td>Specifies a name whose value will be used in place of the text option resource.</td>
</tr>
<tr>
<td>underline</td>
<td>If set, specifies the index (0-based) of a character to underline in the text string. The underline character is used for mnemonic activation.</td>
</tr>
<tr>
<td>image</td>
<td>Specifies an image to display. This is a list of 1 or more elements. The first element is the default image name. The rest of the list is a sequence of statespec/value pairs as defined by <code>Style.map()</code>, specifying different images to use when the widget is in a particular state or a combination of states. All images in the list should have the same</td>
</tr>
<tr>
<td>option</td>
<td>description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>state</td>
<td>May be set to “normal” or “disabled” to control the “disabled” state bit. This is a write-only option: setting it changes the widget state, but the \texttt{Widget.state()} method does not affect this option.</td>
</tr>
</tbody>
</table>

### 24.2.3.4. Compatibility Options

### 24.2.3.5. Widget States

The widget state is a bitmap of independent state flags.

<table>
<thead>
<tr>
<th>flag</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>active</td>
<td>The mouse cursor is over the widget and pressing a mouse button will cause some action to occur.</td>
</tr>
<tr>
<td>disabled</td>
<td>Widget is disabled under program control.</td>
</tr>
<tr>
<td>focus</td>
<td>Widget has keyboard focus.</td>
</tr>
<tr>
<td>State</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>pressed</td>
<td>Widget is being pressed.</td>
</tr>
<tr>
<td>selected</td>
<td>“On”, “true”, or “current” for things like Checkbuttons and radiobuttons.</td>
</tr>
<tr>
<td>background</td>
<td>Windows and Mac have a notion of an “active” or foreground window. The <em>background</em> state is set for widgets in a background window, and cleared for those in the foreground window.</td>
</tr>
<tr>
<td>readonly</td>
<td>Widget should not allow user modification.</td>
</tr>
<tr>
<td>alternate</td>
<td>A widget-specific alternate user display format.</td>
</tr>
<tr>
<td>invalid</td>
<td>The widget’s value is invalid.</td>
</tr>
</tbody>
</table>

A state specification is a sequence of state names, optionally prefixed with an exclamation point indicating that the bit is off.

### 24.2.3.6. ttk.Widget

Besides the methods described below, the `ttk.Widget` class supports the `Tkinter.Widget.cget()` and `Tkinter.Widget.configure()` methods.

```
class ttk.Widget

identify(x, y)

Returns the name of the element at position x y, or the empty string if the point does not lie within any element.

x and y are pixel coordinates relative to the widget.

instate(statespec, callback=None, *args, **kw)

Test the widget’s state. If a callback is not specified, returns `True` if the widget state matches `statespec` and `False` otherwise. If callback is specified then it is called with `args` if widget state matches `statespec`.

state([statespec=None])
```
Modify or read widget state. If `statespec` is specified, sets the widget state accordingly and returns a new `statespec` indicating which flags were changed. If `statespec` is not specified, returns the currently-enabled state flags.

`statespec` will usually be a list or a tuple.
24.2.4. Combobox

The `ttk.Combobox` widget combines a text field with a pop-down list of values. This widget is a subclass of `Entry`.

Besides the methods inherited from `Widget` (`Widget.cget()`, `Widget.configure()`, `Widget.identify()`, `Widget.instate()`, and `Widget.state()`), and those inherited from `Entry` (`Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.selection()`, `Entry.xview()`), this class has some other methods, described at `ttk.Combobox`.

24.2.4.1. Options

This widget accepts the following options:

<table>
<thead>
<tr>
<th>option</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exportselection</td>
<td>Boolean value. If set, the widget selection is linked to the Window Manager selection (which can be returned by invoking <code>Misc.selection_get()</code>, for example).</td>
</tr>
<tr>
<td>justify</td>
<td>Specifies how the text is aligned within the widget. One of “left”, “center”, or “right”.</td>
</tr>
<tr>
<td>height</td>
<td>Specifies the height of the pop-down listbox, in rows.</td>
</tr>
<tr>
<td>postcommand</td>
<td>A script (possibly registered with <code>Misc.register()</code>) that is called immediately before displaying the values. It may specify which values to display.</td>
</tr>
<tr>
<td>state</td>
<td>One of “normal”, “readonly”, or “disabled”. In the “readonly” state, the value may not be edited directly, and the user can only select one of the values from the dropdown list.</td>
</tr>
</tbody>
</table>
the “normal” state, the text field is directly editable. In the “disabled” state, no interaction is possible.

| textvariable | Specifies a name whose value is linked to the widget value. Whenever the value associated with that name changes, the widget value is updated, and vice versa. See Tkinter.StringVar. |
| values | Specifies the list of values to display in the drop-down listbox. |
| width | Specifies an integer value indicating the desired width of the entry window, in average-size characters of the widget’s font. |

24.2.4.2. Virtual events

The combobox widget generates a <<ComboboxSelected>> virtual event when the user selects an element from the list of values.

24.2.4.3. ttk.Combobox

class ttk.Combobox

    current([newindex=None])

        If newindex is specified, sets the combobox value to the element position newindex. Otherwise, returns the index of the current value or -1 if the current value is not in the values list.

    get()

        Returns the current value of the combobox.

    set(value)

        Sets the value of the combobox to value.
24.2.5. Notebook

The Ttk Notebook widget manages a collection of windows and displays a single one at a time. Each child window is associated with a tab, which the user may select to change the currently-displayed window.

24.2.5.1. Options

This widget accepts the following specific options:

<table>
<thead>
<tr>
<th>option</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
<td>If present and greater than zero, specifies the desired height of the pane area (not including internal padding or tabs). Otherwise, the maximum height of all panes is used.</td>
</tr>
<tr>
<td>padding</td>
<td>Specifies the amount of extra space to add around the outside of the notebook. The padding is a list of up to four length specifications: left top right bottom. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left.</td>
</tr>
<tr>
<td>width</td>
<td>If present and greater than zero, specifies the desired width of the pane area (not including internal padding). Otherwise, the maximum width of all panes is used.</td>
</tr>
</tbody>
</table>

24.2.5.2. Tab Options

There are also specific options for tabs:

<table>
<thead>
<tr>
<th>option</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>Either “normal”, “disabled” or “hidden”. If “disabled”, then the tab is not selectable. If</td>
</tr>
<tr>
<td><strong>sticky</strong></td>
<td>Specifies how the child window is positioned within the pane area. Value is a string containing zero or more of the characters “n”, “s”, “e” or “w”. Each letter refers to a side (north, south, east or west) that the child window will stick to, as per the <code>grid()</code> geometry manager.</td>
</tr>
<tr>
<td><strong>padding</strong></td>
<td>Specifies the amount of extra space to add between the notebook and this pane. Syntax is the same as for the option padding used by this widget.</td>
</tr>
<tr>
<td><strong>text</strong></td>
<td>Specifies a text to be displayed in the tab.</td>
</tr>
<tr>
<td><strong>image</strong></td>
<td>Specifies an image to display in the tab. See the option image described in <code>Widget</code>.</td>
</tr>
<tr>
<td><strong>compound</strong></td>
<td>Specifies how to display the image relative to the text, in the case both text and image options are present. See <code>Label Options</code> for legal values.</td>
</tr>
<tr>
<td><strong>underline</strong></td>
<td>Specifies the index (0-based) of a character to underline in the text string. The underlined character is used for mnemonic activation if <code>Notebook.enable_traversal()</code> is called.</td>
</tr>
</tbody>
</table>

### 24.2.5.3. Tab Identifiers

The *tab_id* present in several methods of `ttk.Notebook` may take any of the following forms:

- An integer between zero and the number of tabs.
- The name of a child window.
- A positional specification of the form “@x,y”, which identifies the tab.
- The literal string “current”, which identifies the currently-selected tab.
- The literal string “end”, which returns the number of tabs (only valid for `Notebook.index()`).
24.2.5.4. Virtual Events

This widget generates a `<NotebookTabChanged>` virtual event after a new tab is selected.

24.2.5.5. ttk.Notebook

class ttk.Notebook

    add(child, **kw)

    Adds a new tab to the notebook.
    
    If window is currently managed by the notebook but hidden, it is restored to its previous position.
    
    See Tab Options for the list of available options.

    forget(tab_id)

    Removes the tab specified by `tab_id`, unmaps and unmanages the associated window.

    hide(tab_id)

    Hides the tab specified by `tab_id`.
    
    The tab will not be displayed, but the associated window remains managed by the notebook and its configuration remembered. Hidden tabs may be restored with the `add()` command.

    identify(x, y)

    Returns the name of the tab element at position `x, y`, or the empty string if none.

    index(tab_id)

    Returns the numeric index of the tab specified by `tab_id`, or the total number of tabs if `tab_id` is the string “end”.
**insert**(*pos, child, **kw*)

Inserts a pane at the specified position.

*pos* is either the string “end”, an integer index, or the name of a managed child. If *child* is already managed by the notebook, moves it to the specified position.

See Tab Options for the list of available options.

**select**(*[tab_id]*)

Selects the specified *tab_id*.

The associated child window will be displayed, and the previously-selected window (if different) is unmapped. If *tab_id* is omitted, returns the widget name of the currently selected pane.

**tab**(*tab_id, option=None, **kw*)

Query or modify the options of the specific *tab_id*.

If *kw* is not given, returns a dictionary of the tab option values. If *option* is specified, returns the value of that *option*. Otherwise, sets the options to the corresponding values.

**tabs**()

Returns a list of windows managed by the notebook.

**enable_traversal**()

Enable keyboard traversal for a toplevel window containing this notebook.

This will extend the bindings for the toplevel window containing the notebook as follows:

- **Control-Tab**: selects the tab following the currently
selected one.

- **Shift-Control-Tab**: selects the tab preceding the currently selected one.
- **Alt-K**: where K is the mnemonic (underlined) character of any tab, will select that tab.

Multiple notebooks in a single toplevel may be enabled for traversal, including nested notebooks. However, notebook traversal only works properly if all panes have the notebook they are in as master.
24.2.6. Progressbar

The `ttk.Progressbar` widget shows the status of a long-running operation. It can operate in two modes: determinate mode shows the amount completed relative to the total amount of work to be done, and indeterminate mode provides an animated display to let the user know that something is happening.

24.2.6.1. Options

This widget accepts the following specific options:

<table>
<thead>
<tr>
<th>option</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orient</td>
<td>One of “horizontal” or “vertical”. Specifies the orientation of the progress bar.</td>
</tr>
<tr>
<td>length</td>
<td>Specifies the length of the long axis of the progress bar (width if horizontal, height if vertical).</td>
</tr>
<tr>
<td>mode</td>
<td>One of “determinate” or “indeterminate”.</td>
</tr>
<tr>
<td>maximum</td>
<td>A number specifying the maximum value. Defaults to 100.</td>
</tr>
<tr>
<td>value</td>
<td>The current value of the progress bar. In “determinate” mode, this represents the amount of work completed. In “indeterminate” mode, it is interpreted as modulo maximum; that is, the progress bar completes one “cycle” when its value increases by maximum.</td>
</tr>
<tr>
<td>variable</td>
<td>A name which is linked to the option value. If specified, the value of the progress bar is automatically set to the value of this name whenever the latter is modified.</td>
</tr>
<tr>
<td>phase</td>
<td>Read-only option. The widget periodically increments the value of this option whenever its value is greater than 0 and, in determinate mode, less than maximum. This option may be used by the</td>
</tr>
</tbody>
</table>
current theme to provide additional animation effects.

# 24.2.6.2. ttk.Progressbar

```python
class ttk.Progressbar

start([interval])
    Begin autoincrement mode: schedules a recurring timer event
    that calls `Progressbar.step()` every `interval` milliseconds. If
    omitted, `interval` defaults to 50 milliseconds.

step([amount])
    Increments the progress bar's value by `amount`.

    `amount` defaults to 1.0 if omitted.

stop()
    Stop autoincrement mode: cancels any recurring timer event
    initiated by `Progressbar.start()` for this progress bar.
```
24.2.7. Separator

The `ttk.Separator` widget displays a horizontal or vertical separator bar.

It has no other methods besides the ones inherited from `ttk.Widget`.

24.2.7.1. Options

This widget accepts the following specific option:

<table>
<thead>
<tr>
<th>option</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orient</td>
<td>One of “horizontal” or “vertical”. Specifies the orientation of the separator.</td>
</tr>
</tbody>
</table>
24.2.8. Sizegrip

The `ttk.sizegrip` widget (also known as a grow box) allows the user to resize the containing toplevel window by pressing and dragging the grip.

This widget has neither specific options nor specific methods, besides the ones inherited from `ttk.Widget`.

24.2.8.1. Platform-specific notes

- On Mac OS X, toplevel windows automatically include a built-in size grip by default. Adding a `sizegrip` is harmless, since the built-in grip will just mask the widget.

24.2.8.2. Bugs

- If the containing toplevel's position was specified relative to the right or bottom of the screen (e.g. ....), the `sizegrip` widget will not resize the window.
- This widget supports only “southeast” resizing.
24.2.9. Treeview

The ttk.Treeview widget displays a hierarchical collection of items. Each item has a textual label, an optional image, and an optional list of data values. The data values are displayed in successive columns after the tree label.

The order in which data values are displayed may be controlled by setting the widget option `displaycolumns`. The tree widget can also display column headings. Columns may be accessed by number or symbolic names listed in the widget option columns. See Column Identifiers.

Each item is identified by a unique name. The widget will generate item IDs if they are not supplied by the caller. There is a distinguished root item, named {}. The root item itself is not displayed; its children appear at the top level of the hierarchy.

Each item also has a list of tags, which can be used to associate event bindings with individual items and control the appearance of the item.

The Treeview widget supports horizontal and vertical scrolling, according to the options described in Scrollable Widget Options and the methods `Treeview.xview()` and `Treeview.yview()`.

24.2.9.1. Options

This widget accepts the following specific options:

<table>
<thead>
<tr>
<th>option</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>columns</td>
<td>A list of column identifiers, specifying the number of columns and their names.</td>
</tr>
<tr>
<td><strong>displaycolumns</strong></td>
<td>A list of column identifiers (either symbolic or integer indices) specifying which data columns are displayed and the order in which they appear, or the string “#all”.</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>height</strong></td>
<td>Specifies the number of rows which should be visible. Note: the requested width is determined from the sum of the column widths.</td>
</tr>
<tr>
<td><strong>padding</strong></td>
<td>Specifies the internal padding for the widget. The padding is a list of up to four length specifications.</td>
</tr>
<tr>
<td><strong>selectmode</strong></td>
<td>Controls how the built-in class bindings manage the selection. One of “extended”, “browse” or “none”. If set to “extended” (the default), multiple items may be selected. If “browse”, only a single item will be selected at a time. If “none”, the selection will not be changed. Note that the application code and tag bindings can set the selection however they wish, regardless of the value of this option.</td>
</tr>
</tbody>
</table>
| **show**          | A list containing zero or more of the following values, specifying which elements of the tree to display.  
  - tree: display tree labels in column #0.  
  - headings: display the heading row.  
  The default is “tree headings”, i.e., show all elements.  
  **Note:** Column #0 always refers to the tree column, even if show=”tree” is not specified. |
24.2.9.2. Item Options

The following item options may be specified for items in the insert and item widget commands.

<table>
<thead>
<tr>
<th>option</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>The textual label to display for the item.</td>
</tr>
<tr>
<td>image</td>
<td>A Tk Image, displayed to the left of the label.</td>
</tr>
<tr>
<td>values</td>
<td>The list of values associated with the item. Each item should have the same number of values as the widget option columns. If there are fewer values than columns, the remaining values are assumed empty. If there are more values than columns, the extra values are ignored.</td>
</tr>
<tr>
<td>open</td>
<td>True/False value indicating whether the item's children should be displayed or hidden.</td>
</tr>
<tr>
<td>tags</td>
<td>A list of tags associated with this item.</td>
</tr>
</tbody>
</table>

24.2.9.3. Tag Options

The following options may be specified on tags:

<table>
<thead>
<tr>
<th>option</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>foreground</td>
<td>Specifies the text foreground color.</td>
</tr>
<tr>
<td>background</td>
<td>Specifies the cell or item background color.</td>
</tr>
<tr>
<td>font</td>
<td>Specifies the font to use when drawing text.</td>
</tr>
<tr>
<td>image</td>
<td>Specifies the item image, in case the item's image option is empty.</td>
</tr>
</tbody>
</table>

24.2.9.4. Column Identifiers

Column identifiers take any of the following forms:
• A symbolic name from the list of columns option.
• An integer n, specifying the nth data column.
• A string of the form #n, where n is an integer, specifying the nth display column.

Notes:

• Item's option values may be displayed in a different order than the order in which they are stored.
• Column #0 always refers to the tree column, even if show="tree" is not specified.

A data column number is an index into an item's option values list; a display column number is the column number in the tree where the values are displayed. Tree labels are displayed in column #0. If option displaycolumns is not set, then data column n is displayed in column #n+1. Again, column #0 always refers to the tree column.

24.2.9.5. Virtual Events

The Treeview widget generates the following virtual events.

<table>
<thead>
<tr>
<th>event</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;TreeviewSelect&gt;&gt;</td>
<td>Generated whenever the selection changes.</td>
</tr>
<tr>
<td>&lt;&lt;TreeviewOpen&gt;&gt;</td>
<td>Generated just before settings the focus item to open=True.</td>
</tr>
<tr>
<td>&lt;&lt;TreeviewClose&gt;&gt;</td>
<td>Generated just after setting the focus item to open=False.</td>
</tr>
</tbody>
</table>

The Treeview.focus() and Treeview.selection() methods can be used to determine the affected item or items.

24.2.9.6. ttk.Treeview
class ttk.Treeview

bbox(item, column=None)
Returns the bounding box (relative to the treeview widget’s window) of the specified item in the form (x, y, width, height).

If column is specified, returns the bounding box of that cell. If the item is not visible (i.e., if it is a descendant of a closed item or is scrolled offscreen), returns an empty string.

get_children([item])
Returns the list of children belonging to item.

If item is not specified, returns root children.

set_children(item, *newchildren)
Replaces item’s child with newchildren.

Children present in item that are not present in newchildren are detached from the tree. No items in newchildren may be an ancestor of item. Note that not specifying newchildren results in detaching item’s children.

column(column, option=None, **kw)
Query or modify the options for the specified column.

If kw is not given, returns a dict of the column option values. If option is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- id
  Returns the column name. This is a read-only option.

- anchor: One of the standard Tk anchor values.
Specifies how the text in this column should be aligned with respect to the cell.

- **minwidth**: width
  The minimum width of the column in pixels. The treeview widget will not make the column any smaller than specified by this option when the widget is resized or the user drags a column.

- **stretch**: True/False
  Specifies whether the column’s width should be adjusted when the widget is resized.

- **width**: width
  The width of the column in pixels.

To configure the tree column, call this with column = “#0”

**delete**(*items*)
Delete all specified *items* and all their descendants.

The root item may not be deleted.

**detach**(*items*)
Unlinks all of the specified *items* from the tree.

The items and all of their descendants are still present, and may be reinserted at another point in the tree, but will not be displayed.

The root item may not be detached.

**exists**(*item*)
Returns **True** if the specified *item* is present in the tree.

**focus**(*item=None*)
If `item` is specified, sets the focus item to `item`. Otherwise, returns the current focus item, or '' if there is none.

**heading**(column, option=None, **kw)

Query or modify the heading options for the specified `column`.

If `kw` is not given, returns a dict of the heading option values. If `option` is specified then the value for that `option` is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- **text**: text
  The text to display in the column heading.

- **image**: imageName
  Specifies an image to display to the right of the column heading.

- **anchor**: anchor
  Specifies how the heading text should be aligned. One of the standard Tk anchor values.

- **command**: callback
  A callback to be invoked when the heading label is pressed.

To configure the tree column heading, call this with column = “#0”.

**identify**(component, x, y)

Returns a description of the specified `component` under the point given by `x` and `y`, or the empty string if no such `component` is present at that position.

**identify_row**(y)
Returns the item ID of the item at position $y$.

**identify_column($x$)**

Returns the data column identifier of the cell at position $x$.

The tree column has ID #0.

**identify_region($x, y$)**

Returns one of:

<table>
<thead>
<tr>
<th>region</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>heading</td>
<td>Tree heading area.</td>
</tr>
<tr>
<td>separator</td>
<td>Space between two columns headings.</td>
</tr>
<tr>
<td>tree</td>
<td>The tree area.</td>
</tr>
<tr>
<td>cell</td>
<td>A data cell.</td>
</tr>
</tbody>
</table>

Availability: Tk 8.6.

**identify_element($x, y$)**

Returns the element at position $x, y$.

Availability: Tk 8.6.

**index($item$)**

Returns the integer index of $item$ within its parent's list of children.

**insert($parent, index, iid=None, **kw$)**

Creates a new item and returns the item identifier of the newly created item.

$parent$ is the item ID of the parent item, or the empty string to create a new top-level item. $index$ is an integer, or the value “end”, specifying where in the list of parent's children to insert the new item. If $index$ is less than or equal to zero, the new
node is inserted at the beginning; if \( index \) is greater than or equal to the current number of children, it is inserted at the end. If \( iid \) is specified, it is used as the item identifier; \( iid \) must not already exist in the tree. Otherwise, a new unique identifier is generated.

See Item Options for the list of available points.

\( \text{item}(item[, \text{option}[[, **kw]])} \)
Query or modify the options for the specified \( item \).

If no options are given, a dict with options/values for the item is returned. If \( \text{option} \) is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values as given by \( kw \).

\( \text{move}(item, parent, index) \)
Moves \( item \) to position \( index \) in \( parent \)'s list of children.

It is illegal to move an item under one of its descendants. If \( index \) is less than or equal to zero, \( item \) is moved to the beginning; if greater than or equal to the number of children, it is moved to the end. If \( item \) was detached it is reattached.

\( \text{next}(item) \)
Returns the identifier of \( item \)'s next sibling, or " if \( item \) is the last child of its parent.

\( \text{parent}(item) \)
Returns the ID of the parent of \( item \), or " if \( item \) is at the top level of the hierarchy.

\( \text{prev}(item) \)
Returns the identifier of \( item \)'s previous sibling, or " if \( item \) is the first child of its parent.
reattach(item, parent, index)
    An alias for Treeview.move().

see(item)
    Ensure that item is visible.

    Sets all of item's ancestors open option to True, and scrolls
    the widget if necessary so that item is within the visible portion
    of the tree.

selection([selop=None, items=None])
    If selop is not specified, returns selected items. Otherwise, it
    will act according to the following selection methods.

selection_set(items)
    items becomes the new selection.

selection_add(items)
    Add items to the selection.

selection_remove(items)
    Remove items from the selection.

selection_toggle(items)
    Toggle the selection state of each item in items.

set(item, column=None, value=None)
    With one argument, returns a dictionary of column/value pairs
    for the specified item. With two arguments, returns the current
    value of the specified column. With three arguments, sets the
    value of given column in given item to the specified value.

tag_bind(tagname, sequence=None, callback=None)
    Bind a callback for the given event sequence to the tag
    tagname. When an event is delivered to an item, the callbacks
for each of the item’s tags option are called.

**tag_configure**(tagname, option=None, **kw)
Query or modify the options for the specified *tagname*.

If *kw* is not given, returns a dict of the option settings for *tagname*. If *option* is specified, returns the value for that *option* for the specified *tagname*. Otherwise, sets the options to the corresponding values for the given *tagname*.

**tag_has**(tagname[, item])
If *item* is specified, returns 1 or 0 depending on whether the specified *item* has the given *tagname*. Otherwise, returns a list of all items that have the specified tag.

*Availability: Tk 8.6*

**xview**(args)
Query or modify horizontal position of the treeview.

**yview**(args)
Query or modify vertical position of the treeview.
Each widget in ttk is assigned a style, which specifies the set of elements making up the widget and how they are arranged, along with dynamic and default settings for element options. By default the style name is the same as the widget’s class name, but it may be overridden by the widget’s style option. If the class name of a widget is unknown, use the method Misc.winfo_class(somewidget.winfo_class()).

**See also:**

Tcl’2004 conference presentation

This document explains how the theme engine works

```python
class ttk.Style
    This class is used to manipulate the style database.

configure(style, query_opt=None, **kw)
    Query or set the default value of the specified option(s) in style.

    Each key in kw is an option and each value is a string identifying the value for that option.

    For example, to change every default button to be a flat button with some padding and a different background color do:

    import ttk
    import Tkinter

    root = Tkinter.Tk()```
```python
ttk.Style().configure("TButton", padding=6, relief="flat"
background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

map(style, query_opt=None, **kw)

Query or sets dynamic values of the specified option(s) in style.

Each key in kw is an option and each value should be a list or a tuple (usually) containing statespecs grouped in tuples, lists, or something else of your preference. A statespec is a compound of one or more states and then a value.

An example:

```python
import Tkinter
import ttk

root = Tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
    background=[('pressed', '!disabled', 'black'), ('active',

colored_btn = ttk.Button(text="Test", style="C.TButton")

root.mainloop()
```

Note that the order of the (states, value) sequences for an option matters. In the previous example, if you change the order to [("active", "blue"), ("pressed", "red")]) in the foreground option, for example, you would get a blue foreground when the widget is in the active or pressed states.
**lookup**(*style*, *option*, *state=None*, *default=None*)

Returns the value specified for *option* in *style*.

If *state* is specified, it is expected to be a sequence of one or more states. If the *default* argument is set, it is used as a fallback value in case no specification for *option* is found.

To check what font a Button uses by default, do:

```python
import ttk
print ttk.Style().lookup("TButton", "font")
```

**layout**(*style*, *layoutspec=None*)

Define the widget layout for given *style*. If *layoutspec* is omitted, return the layout specification for given *style*.

*layoutspec*, if specified, is expected to be a list or some other sequence type (excluding strings), where each item should be a tuple and the first item is the layout name and the second item should have the format described in Layouts.

To understand the format, see the following example (it is not intended to do anything useful):

```python
import ttk
import Tkinter

root = Tkinter.Tk()

style = ttk.Style()
style.layout("TMenubutton", [
    ("Menubutton.background", None),
    ("Menubutton.button", {"children":
        [("Menubutton.focus", {"children":
            [("Menubutton.padding", {"children":
                [("Menubutton.label", {"side": "left", "ex
            }])
        }]
    }])
})
]}
```
```python
mbtn = ttk.Menubutton(text='Text')
mbtn.pack()
root.mainloop()
```

**element_create(elementname, etype, *args, **kw)**

Create a new element in the current theme, of the given `etype` which is expected to be either “image”, “from” or “vsapi”. The latter is only available in Tk 8.6a for Windows XP and Vista and is not described here.

If “image” is used, `args` should contain the default image name followed by statespec/value pairs (this is the imagespec), and `kw` may have the following options:

- **border=padding**
  padding is a list of up to four integers, specifying the left, top, right, and bottom borders, respectively.

- **height=height**
  Specifies a minimum height for the element. If less than zero, the base image’s height is used as a default.

- **padding=padding**
  Specifies the element’s interior padding. Defaults to border’s value if not specified.

- **sticky=spec**
  Specifies how the image is placed within the final parcel. spec contains zero or more characters “n”, “s”, “w”, or “e”.

- **width=width**
Specifies a minimum width for the element. If less than zero, the base image's width is used as a default.

If “from” is used as the value of etype, element_create() will clone an existing element. args is expected to contain a themename, from which the element will be cloned, and optionally an element to clone from. If this element to clone from is not specified, an empty element will be used. kw is discarded.

element_names()
Returns the list of elements defined in the current theme.

element_options(elementname)
Returns the list of elementname's options.

theme_create(themename, parent=None, settings=None)
Create a new theme.

It is an error if themename already exists. If parent is specified, the new theme will inherit styles, elements and layouts from the parent theme. If settings are present they are expected to have the same syntax used for theme_settings().

theme_settings(themename, settings)
Temporarily sets the current theme to themename, apply specified settings and then restore the previous theme.

Each key in settings is a style and each value may contain the keys ‘configure’, ‘map’, ‘layout’ and ‘element create’ and they are expected to have the same format as specified by the methods Style.configure(), Style.map(), Style.layout() and Style.element_create() respectively.
As an example, let's change the Combobox for the default theme a bit:

```python
import ttk
import Tkinter

root = Tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
    "TCombobox": {
        "configure": {"padding": 5},
        "map": {
            "background": ["active", "green2"],
            (!disabled", "green4"],
            "fieldbackground": ["!disabled", "green3"],
            "foreground": ["!disabled", "OliveDrab2"]
        }
    }
})

combo = ttk.Combobox().pack()

root.mainloop()
```

`theme_names()`

Returns a list of all known themes.

`theme_use([themename])`

If `themename` is not given, returns the theme in use. Otherwise, sets the current theme to `themename`, refreshes all widgets and emits a `<<ThemeChanged>>` event.

### 24.2.10.1. Layouts

A layout can be just `None`, if it takes no options, or a dict of options specifying how to arrange the element. The layout mechanism uses a simplified version of the pack geometry manager: given an initial cavity, each element is allocated a parcel. Valid options/values are:
- side: whichside
  Specifies which side of the cavity to place the element; one of top, right, bottom or left. If omitted, the element occupies the entire cavity.

- sticky: nswe
  Specifies where the element is placed inside its allocated parcel.

- unit: 0 or 1
  If set to 1, causes the element and all of its descendants to be treated as a single element for the purposes of `Widget.identify()` et al. It's used for things like scrollbar thumbs with grips.

- children: [sublayout...]
  Specifies a list of elements to place inside the element. Each element is a tuple (or other sequence type) where the first item is the layout name, and the other is a Layout.
24.3. **Tix — Extension widgets for Tk**

The **Tix** (Tk Interface Extension) module provides an additional rich set of widgets. Although the standard Tk library has many useful widgets, they are far from complete. The **Tix** library provides most of the commonly needed widgets that are missing from standard Tk: **HList, ComboBox, Control** (a.k.a. SpinBox) and an assortment of scrollable widgets. **Tix** also includes many more widgets that are generally useful in a wide range of applications: **NoteBook, FileEntry, PanedWindow**, etc; there are more than 40 of them.

With all these new widgets, you can introduce new interaction techniques into applications, creating more useful and more intuitive user interfaces. You can design your application by choosing the most appropriate widgets to match the special needs of your application and users.

**Note:** **Tix** has been renamed to **tkinter.tix** in Python 3. The **2to3** tool will automatically adapt imports when converting your sources to Python 3.

**See also:**

**Tix Homepage**
The home page for **Tix**. This includes links to additional documentation and downloads.

**Tix Man Pages**
On-line version of the man pages and reference material.

**Tix Programming Guide**
On-line version of the programmer’s reference material.
**Tix Development Applications**

Tix applications for development of Tix and Tkinter programs. Tide applications work under Tk or Tkinter, and include **TixInspect**, an inspector to remotely modify and debug Tix/Tk/Tkinter applications.
24.3.1. Using Tix

`class Tix.Tix(screenName[, baseName[, className]])`

Toplevel widget of Tix which represents mostly the main window of an application. It has an associated Tcl interpreter.

Classes in the Tix module subclasses the classes in the Tkinter module. The former imports the latter, so to use Tix with Tkinter, all you need to do is to import one module. In general, you can just import Tix, and replace the toplevel call to Tkinter.Tk with Tix.Tk:

```python
import Tix from Tkconstants import *
root = Tix.Tk()
```

To use Tix, you must have the Tix widgets installed, usually alongside your installation of the Tk widgets. To test your installation, try the following:

```python
import Tix
root = Tix.Tk()
root.tk.eval('package require Tix')
```

If this fails, you have a Tk installation problem which must be resolved before proceeding. Use the environment variable TIX_LIBRARY to point to the installed Tix library directory, and make sure you have the dynamic object library (tix8183.dll or libtix8183.so) in the same directory that contains your Tk dynamic object library (tk8183.dll or libtk8183.so). The directory with the dynamic object library should also have a file called pkgIndex.tcl (case sensitive), which contains the line:
package ifneeded Tix 8.1 [list load "file join $dir tix8183.dll"]
24.3.2. Tix Widgets

Tix introduces over 40 widget classes to the Tkinter repertoire. There is a demo of all the Tix widgets in the demo/tix directory of the standard distribution.

24.3.2.1. Basic Widgets

*class* Tix.

**Balloon**

A Balloon that pops up over a widget to provide help. When the user moves the cursor inside a widget to which a Balloon widget has been bound, a small pop-up window with a descriptive message will be shown on the screen.

*class* Tix.

**ButtonBox**

The ButtonBox widget creates a box of buttons, such as is commonly used for Ok Cancel.

*class* Tix.

**ComboBox**

The ComboBox widget is similar to the combo box control in MS Windows. The user can select a choice by either typing in the entry subwidget or selecting from the listbox subwidget.

*class* Tix.

**Control**

The Control widget is also known as the SpinBox widget. The user can adjust the value by pressing the two arrow buttons or by entering the value directly into the entry. The new value will be checked against the user-defined upper and lower limits.

*class* Tix.

**LabelEntry**

The LabelEntry widget packages an entry widget and a label into one mega widget. It can be used to simplify the creation of “entry-form” type of interface.
class Tix.LabelFrame
The LabelFrame widget packages a frame widget and a label into one mega widget. To create widgets inside a LabelFrame widget, one creates the new widgets relative to the frame subwidget and manage them inside the frame subwidget.

class Tix.Meter
The Meter widget can be used to show the progress of a background job which may take a long time to execute.

class Tix.OptionMenu
The OptionMenu creates a menu button of options.

class Tix.PopupMenu
The PopupMenu widget can be used as a replacement of the tk_popup command. The advantage of the Tix PopupMenu widget is it requires less application code to manipulate.

class Tix.Select
The Select widget is a container of button subwidgets. It can be used to provide radio-box or check-box style of selection options for the user.

class Tix.StdButtonBox
The StdButtonBox widget is a group of standard buttons for Motif-like dialog boxes.

24.3.2.2. File Selectors

class Tix.DirList
The DirList widget displays a list view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class Tix.DirTree
The DirTree widget displays a tree view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class Tix.DirSelectDialog
The DirSelectDialog widget presents the directories in the file system in a dialog window. The user can use this dialog window to navigate through the file system to select the desired directory.

class Tix.DirSelectBox
The DirSelectBox is similar to the standard Motif(TM) directory-selection box. It is generally used for the user to choose a directory. DirSelectBox stores the directories mostly recently selected into a ComboBox widget so that they can be quickly selected again.

class Tix.ExFileSelectBox
The ExFileSelectBox widget is usually embedded in a tixExFileSelectDialog widget. It provides a convenient method for the user to select files. The style of the ExFileSelectBox widget is very similar to the standard file dialog on MS Windows 3.1.

class Tix.FileSelectBox
The FileSelectBox is similar to the standard Motif(TM) file-selection box. It is generally used for the user to choose a file. FileSelectBox stores the files mostly recently selected into a ComboBox widget so that they can be quickly selected again.

class Tix.FileEntry
The FileEntry widget can be used to input a filename. The user can type in the filename manually. Alternatively, the user can press the button widget that sits next to the entry, which will bring up a file selection dialog.
24.3.2.3. Hierarchical ListBox

*class Tix.HList*

The HList widget can be used to display any data that have a hierarchical structure, for example, file system directory trees. The list entries are indented and connected by branch lines according to their places in the hierarchy.

*class Tix.CheckList*

The CheckList widget displays a list of items to be selected by the user. CheckList acts similarly to the Tk checkbutton or radiobutton widgets, except it is capable of handling many more items than checkbuttons or radiobuttons.

*class Tix.Tree*

The Tree widget can be used to display hierarchical data in a tree form. The user can adjust the view of the tree by opening or closing parts of the tree.

24.3.2.4. Tabular ListBox

*class Tix.TList*

The TList widget can be used to display data in a tabular format. The list entries of a TList widget are similar to the entries in the Tk listbox widget. The main differences are (1) the TList widget can display the list entries in a two dimensional format and (2) you can use graphical images as well as multiple colors and fonts for the list entries.

24.3.2.5. Manager Widgets

*class Tix.PanedWindow*

The PanedWindow widget allows the user to interactively manipulate the sizes of several panes. The panes can be
arranged either vertically or horizontally. The user changes the sizes of the panes by dragging the resize handle between two panes.

**class Tix. ListNoteBook**

The ListNoteBook widget is very similar to the TixNoteBook widget: it can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages (windows). At one time only one of these pages can be shown. The user can navigate through these pages by choosing the name of the desired page in the hlist subwidget.

**class Tix. NoteBook**

The NoteBook widget can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages. At one time only one of these pages can be shown. The user can navigate through these pages by choosing the visual “tabs” at the top of the NoteBook widget.

### 24.3.2.6. Image Types

The Tix module adds:

- **pixmap** capabilities to all Tix and Tkinter widgets to create color images from XPM files.
- **Compound** image types can be used to create images that consists of multiple horizontal lines; each line is composed of a series of items (texts, bitmaps, images or spaces) arranged from left to right. For example, a compound image can be used to display a bitmap and a text string simultaneously in a Tk Button widget.

### 24.3.2.7. Miscellaneous Widgets
**class** Tix. **InputOnly**

The **InputOnly** widgets are to accept inputs from the user, which can be done with the `bind` command (Unix only).

### 24.3.2.8. Form Geometry Manager

In addition, Tix augments Tkinter by providing:

**class** Tix. **Form**

The **Form** geometry manager based on attachment rules for all Tk widgets.
class Tix.tixCommand

The \texttt{tix} \texttt{commands} provide access to miscellaneous elements of \texttt{Tix}'s internal state and the \texttt{Tix} application context. Most of the information manipulated by these methods pertains to the application as a whole, or to a screen or display, rather than to a particular window.

To view the current settings, the common usage is:

```python
import Tix
root = Tix.Tk()
print root.tix_configure()
```

tixCommand.\texttt{tix\_configure}(\texttt{cnf=None **kw})

Query or modify the configuration options of the Tix application context. If no option is specified, returns a dictionary all of the available options. If option is specified with no value, then the method returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the method modifies the given option(s) to have the given value(s); in this case the method returns an empty string. Option may be any of the configuration options.

tixCommand.\texttt{tix\_cget}(\texttt{option})

Returns the current value of the configuration option given by \texttt{option}. Option may be any of the configuration options.

tixCommand.\texttt{tix\_getbitmap}(\texttt{name})

Locates a bitmap file of the name \texttt{name.xpm} or \texttt{name} in one of the bitmap directories (see the \texttt{tix\_addbitmapdir()} method). By using \texttt{tix\_getbitmap()}, you can avoid hard coding the
pathnames of the bitmap files in your application. When successful, it returns the complete pathname of the bitmap file, prefixed with the character @. The returned value can be used to configure the bitmap option of the Tk and Tix widgets.

tixCommand.tix_addbitmapdir(directory)
Tix maintains a list of directories under which the tix_getimage() and tix_getbitmap() methods will search for image files. The standard bitmap directory is $TIX_LIBRARY/bitmaps. The tix_addbitmapdir() method adds directory into this list. By using this method, the image files of an applications can also be located using the tix_getimage() or tix_getbitmap() method.

tixCommand.tix_filedialog([dlgclass])
Returns the file selection dialog that may be shared among different calls from this application. This method will create a file selection dialog widget when it is called the first time. This dialog will be returned by all subsequent calls to tix_filedialog(). An optional dlgclass parameter can be passed as a string to specified what type of file selection dialog widget is desired. Possible options are tix, FileSelectDialog or tixExFileSelectDialog.

tixCommand.tix_getimage(self, name)
Locates an image file of the name name.xpm, name.xbm or name.ppm in one of the bitmap directories (see the tix_addbitmapdir() method above). If more than one file with the same name (but different extensions) exist, then the image type is chosen according to the depth of the X display: xbm images are chosen on monochrome displays and color images are chosen on color displays. By using tix_getimage(), you can avoid hard coding the pathnames of the image files in your application. When successful, this method returns the name of
the newly created image, which can be used to configure the image option of the Tk and Tix widgets.

tixCommand. **tix_option_get**(name)

Gets the options maintained by the Tix scheme mechanism.

tixCommand. **tix_resetoptions**(newScheme, newFontSet[, newScmPrio])

Resets the scheme and fontset of the Tix application to newScheme and newFontSet, respectively. This affects only those widgets created after this call. Therefore, it is best to call the resetoptions method before the creation of any widgets in a Tix application.

The optional parameter newScmPrio can be given to reset the priority level of the Tk options set by the Tix schemes.

Because of the way Tk handles the X option database, after Tix has been has imported and initied, it is not possible to reset the color schemes and font sets using the tix_config() method. Instead, the tix_resetoptions() method must be used.
24.4. `ScrollledText` — Scrolled Text Widget

The `ScrollledText` module provides a class of the same name which implements a basic text widget which has a vertical scroll bar configured to do the “right thing.” Using the `ScrollledText` class is a lot easier than setting up a text widget and scroll bar directly. The constructor is the same as that of the `Tkinter.Text` class.

**Note:** `ScrollledText` has been renamed to `tkinter.scrolledtext` in Python 3. The `2to3` tool will automatically adapt imports when converting your sources to Python 3.

The text widget and scrollbar are packed together in a `Frame`, and the methods of the `Grid` and `Pack` geometry managers are acquired from the `Frame` object. This allows the `ScrollledText` widget to be used directly to achieve most normal geometry management behavior.

Should more specific control be necessary, the following attributes are available:

- `ScrollledText.frame`
  - The frame which surrounds the text and scroll bar widgets.

- `ScrollledText.vbar`
  - The scroll bar widget.
24.5. turtle — Turtle graphics for Tk
24.5.1. Introduction

Turtle graphics is a popular way for introducing programming to kids. It was part of the original Logo programming language developed by Wally Feurzig and Seymour Papert in 1966.

Imagine a robotic turtle starting at (0, 0) in the x-y plane. After an import turtle, give it the command turtle.forward(15), and it moves (on-screen!) 15 pixels in the direction it is facing, drawing a line as it moves. Give it the command turtle.right(25), and it rotates in-place 25 degrees clockwise.

By combining together these and similar commands, intricate shapes and pictures can easily be drawn.

The turtle module is an extended reimplementation of the same-named module from the Python standard distribution up to version Python 2.5.

It tries to keep the merits of the old turtle module and to be (nearly) 100% compatible with it. This means in the first place to enable the learning programmer to use all the commands, classes and methods interactively when using the module from within IDLE run with the -n switch.

The turtle module provides turtle graphics primitives, in both object-oriented and procedure-oriented ways. Because it uses Tkinter for the underlying graphics, it needs a version of Python installed with Tk support.

The object-oriented interface uses essentially two+two classes:

1. The TurtleScreen class defines graphics windows as a playground for the drawing turtles. Its constructor needs a
Tkinter.Canvas or a ScrolledCanvas as argument. It should be used when turtle is used as part of some application.

The function Screen() returns a singleton object of a TurtleScreen subclass. This function should be used when turtle is used as a standalone tool for doing graphics. As a singleton object, inheriting from its class is not possible.

All methods of TurtleScreen/Screen also exist as functions, i.e. as part of the procedure-oriented interface.

2. RawTurtle (alias: RawPen) defines Turtle objects which draw on a TurtleScreen. Its constructor needs a Canvas, ScrolledCanvas or TurtleScreen as argument, so the RawTurtle objects know where to draw.

Derived from RawTurtle is the subclass Turtle (alias: Pen), which draws on “the” Screen - instance which is automatically created, if not already present.

All methods of RawTurtle/Turtle also exist as functions, i.e. part of the procedure-oriented interface.

The procedural interface provides functions which are derived from the methods of the classes Screen and Turtle. They have the same names as the corresponding methods. A screen object is automatically created whenever a function derived from a Screen method is called. An (unnamed) turtle object is automatically created whenever any of the functions derived from a Turtle method is called.

To use multiple turtles an a screen one has to use the object-oriented interface.

Note: In the following documentation the argument list for
functions is given. Methods, of course, have the additional first argument *self* which is omitted here.
24.5.2. Overview over available Turtle and Screen methods

24.5.2.1. Turtle methods

Turtle motion
  Move and draw
    forward() | fd()
    backward() | bk() | back()
    right() | rt()
    left() | lt()
    goto() | setpos() | setposition()
    setx()
    sety()
    setheading() | seth()
    home()
    circle()
    dot()
    stamp()
    clearstamp()
    clearstamps()
    undo()
    speed()

Tell Turtle’s state
  position() | pos()
  towards()
  xcor()
  ycor()
  heading()
  distance()
Setting and measurement

- degrees()
- radians()

Pen control

Drawing state

- pendown() | pd() | down()
- penup() | pu() | up()
- pensize() | width()
- pen()
- isdown()

Color control

- color()
- pencolor()
- fillcolor()

Filling

- fill()
- begin_fill()
- end_fill()

More drawing control

- reset()
- clear()
- write()

Turtle state

Visibility

- showturtle() | st()
- hideturtle() | ht()
- isvisible()

Appearance

- shape()
- resizemode()
shapesize() | turtlesize()
settiltangle()
tiltangle()
tilt()

Using events
onclick()
onrelease()
ondrag()
mainloop() | done()

Special Turtle methods
begin_poly()
end_poly()
get_poly()
clone()
getturtle() | getpen()
getscreen()
setundobuffer()
undobufferentries()
tracer()
window_width()
window_height()

24.5.2.2. Methods of TurtleScreen/Screen

Window control
bgcolor()
bgpic()
clear() | cleargraph()
reset() | resetscreen()
screensize()
setworldcoordinates()

Animation control
delay()
tracer()
update()

Using screen events

listen()
onkey()
onclick() | onscreenclck()
ontimer()

Settings and special methods

mode()
colormode()
g etcanvas()
g etshapes()
register_shape() | addshape()
turtles()
window_height()
window_width()

Methods specific to Screen

bye()
exitonclick()
setup()
title()
24.5.3. Methods of RawTurtle/Turtle and corresponding functions

Most of the examples in this section refer to a Turtle instance called `turtle`.

### 24.5.3.1. Turtle motion

- `turtle.forward(distance)`
- `turtle.fd(distance)`

**Parameters:** `distance` – a number (integer or float)

Move the turtle forward by the specified `distance`, in the direction the turtle is headed.

```python
>>> turtle.position()
(0.00, 0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00, 0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00, 0.00)
```

- `turtle.back(distance)`
- `turtle.bk(distance)`
- `turtle.backward(distance)`

**Parameters:** `distance` – a number

Move the turtle backward by `distance`, opposite to the direction the turtle is headed. Do not change the turtle's heading.

```python
>>> turtle.position()
(0.00, 0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00, 0.00)
```
turtle.right(angle)
turtle.rt(angle)

**Parameters:** angle – a number (integer or float)

Turn turtle right by angle units. (Units are by default degrees, but can be set via the `degrees()` and `radians()` functions.) Angle orientation depends on the turtle mode, see `mode()`.

```python
turtle.right(45)
turtle.heading()
```

turtle.left(angle)
turtle.lt(angle)

**Parameters:** angle – a number (integer or float)

Turn turtle left by angle units. (Units are by default degrees, but can be set via the `degrees()` and `radians()` functions.) Angle orientation depends on the turtle mode, see `mode()`.

```python
turtle.left(45)
turtle.heading()
```

turtle.goto(x, y=None)
turtle.setpos(x, y=None)
turtle.setposition(x, y=None)

**Parameters:**
- x – a number or a pair/vector of numbers
- y – a number or None

If y is `None`, x must be a pair of coordinates or a `Vec2D` (e.g. as returned by `pos()`).

Move turtle to an absolute position. If the pen is down, draw line.
Do not change the turtle's orientation.

```python
>>> tp = turtle.pos()
>>> tp
(0.00, 0.00)
>>> turtle.setpos(60, 30)
>>> turtle.pos()
(60.00, 30.00)
>>> turtle.setpos((20, 80))
>>> turtle.pos()
(20.00, 80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00, 0.00)
```

turtle.setx(x)

**Parameters:** x – a number (integer or float)

Set the turtle's first coordinate to x, leave second coordinate unchanged.

```python
>>> turtle.position()
(0.00, 240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00, 240.00)
```

turtle.sety(y)

**Parameters:** y – a number (integer or float)

Set the turtle's second coordinate to y, leave first coordinate unchanged.

```python
>>> turtle.position()
(0.00, 40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00, -10.00)
```

turtle.setheading(to_angle)
turtle.seth(to_angle)
**Parameters:** to_angle – a number (integer or float)

Set the orientation of the turtle to to_angle. Here are some common directions in degrees:

<table>
<thead>
<tr>
<th>standard mode</th>
<th>logo mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - east</td>
<td>0 - north</td>
</tr>
<tr>
<td>90 - north</td>
<td>90 - east</td>
</tr>
<tr>
<td>180 - west</td>
<td>180 - south</td>
</tr>
<tr>
<td>270 - south</td>
<td>270 - west</td>
</tr>
</tbody>
</table>

```python
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

turtle. home()

Move turtle to the origin – coordinates (0,0) – and set its heading to its start-orientation (which depends on the mode, see mode()).

```python
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00, -10.00)
>>> turtle.home()
>>> turtle.position()
(0.00, 0.00)
>>> turtle.heading()
0.0
```

turtle. circle(radius, extent=None, steps=None)

**Parameters:**
- **radius** – a number
- **extent** – a number (or None)
- **steps** – an integer (or None)

Draw a circle with given radius. The center is radius units left of the turtle; extent – an angle – determines which part of the circle is drawn. If extent is not given, draw the entire circle. If extent is not a full circle, one endpoint of the arc is the current pen
position. Draw the arc in counterclockwise direction if \textit{radius} is positive, otherwise in clockwise direction. Finally the direction of the turtle is changed by the amount of \textit{extent}.

As the circle is approximated by an inscribed regular polygon, \textit{steps} determines the number of steps to use. If not given, it will be calculated automatically. May be used to draw regular polygons.

\begin{verbatim}
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180)  # draw a semicircle
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0
\end{verbatim}

turtle.\texttt{dot}(\texttt{size=}=\texttt{None}, *\texttt{color})

\textbf{Parameters:}
\begin{itemize}
  \item \texttt{size} – an integer $\geq 1$ (if given)
  \item \texttt{color} – a colorstring or a numeric color tuple
\end{itemize}

Draw a circular dot with diameter \texttt{size}, using \texttt{color}. If \texttt{size} is not given, the maximum of pensize+4 and 2*pensize is used.

\begin{verbatim}
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0
\end{verbatim}

turtle.\texttt{stamp}()
Stamp a copy of the turtle shape onto the canvas at the current turtle position. Return a stamp_id for that stamp, which can be used to delete it by calling `clearstamp(stamp_id)`.

```python
>>> turtle.color("blue")
>>> turtle.stamp()
11
>>> turtle.fd(50)
```

turtle. `clearstamp(stampid)`

**Parameters:** `stampid` – an integer, must be return value of previous `stamp()` call

Delete stamp with given `stampid`.

```python
>>> turtle.position()
(150.00, -0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00, -0.00)
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00, -0.00)
```

turtle. `clearstamps(n=None)`

**Parameters:** `n` – an integer (or `None`)

Delete all or first/last `n` of turtle’s stamps. If `n` is `None`, delete all stamps, if `n > 0` delete first `n` stamps, else if `n < 0` delete last `n` stamps.

```python
>>> for i in range(8):
...     turtle.stamp(); turtle.fd(30)
13
14
15
16
17
18
```
turtle.\texttt{undo()}

Undo (repeatedly) the last turtle action(s). Number of available undo actions is determined by the size of the undobuffer.

\begin{verbatim}
>>> for i in range(4):
    ...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
    ...     turtle.undo()
\end{verbatim}

turtle.\texttt{speed(speed=\textit{None})}

\begin{tabular}{|l|}
\hline
\textbf{Parameters}: \\
\hline
\texttt{speed} – an integer in the range 0..10 or a speedstring (see below) \\
\hline
\end{tabular}

Set the turtle’s speed to an integer value in the range 0..10. If no argument is given, return current speed.

If input is a number greater than 10 or smaller than 0.5, speed is set to 0. Speedstrings are mapped to speedvalues as follows:

- “fastest”: 0
- “fast”: 10
- “normal”: 6
- “slow”: 3
- “slowest”: 1

Speeds from 1 to 10 enforce increasingly faster animation of line drawing and turtle turning.

Attention: \texttt{speed = 0} means that \textit{no} animation takes place. forward/back makes turtle jump and likewise left/right make the turtle turn instantly.
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9

24.5.3.2. Tell Turtle’s state

turtle.**position**()
turtle.**pos**()

Return the turtle’s current location (x,y) (as a **Vec2D** vector).

```python
>>> turtle.pos()
(440.00, -0.00)
```

turtle.**towards**(x, y=None)

**Parameters:**

- x – a number or a pair/vector of numbers or a turtle instance
- y – a number if x is a number, else None

Return the angle between the line from turtle position to position specified by (x,y), the vector or the other turtle. This depends on the turtle’s start orientation which depends on the mode - “standard”/”world” or “logo”).

```python
>>> turtle.goto(10, 10)
>>> turtle.towards(0, 0)
225.0
```

turtle.**xcor**()

Return the turtle’s x coordinate.

```python
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
```
turtle.pos()
(64.28, 76.60)
>>> print turtle.xcor()
64.2787609687

turtle.ycor()
Return the turtle's y coordinate.

>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print turtle.pos()
(50.00, 86.60)
>>> print turtle.ycor()
86.6025403784

turtle.heading()
Return the turtle's current heading (value depends on the turtle mode, see mode()).

>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0

turtle.distance(x, y=None)
Parameters:
- x – a number or a pair/vector of numbers or a turtle instance
- y – a number if x is a number, else None

Return the distance from the turtle to (x,y), the given vector, or the given other turtle, in turtle step units.

>>> turtle.home()
>>> turtle.distance(30, 40)
50.0
>>> turtle.distance((30, 40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
turtle.distance(joe)
77.0

24.5.3.3. Settings for measurement

turtle. degrees(fullcircle=360.0)

Parameters: fullcircle – a number

Set angle measurement units, i.e. set number of “degrees” for a full circle. Default value is 360 degrees.

>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

Change angle measurement unit to grad (also known as gon, grade, or gradian and equals 1/100-th of the right angle.)

>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0

turtle. radians()

Set the angle measurement units to radians. Equivalent to degrees(2*math.pi).

>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966

24.5.3.4. Pen control

24.5.3.4.1. Drawing state
turtle. **pendown()**  
turtle. **pd()**  
turtle. **down()**  
Pull the pen down – drawing when moving.

turtle. **penup()**  
turtle. **pu()**  
turtle. **up()**  
Pull the pen up – no drawing when moving.

turtle. **pensize(width=None)**  
turtle. **width(width=None)**  

**Parameters:** width – a positive number

Set the line thickness to *width* or return it. If resizemode is set to “auto” and turtleshape is a polygon, that polygon is drawn with the same line thickness. If no argument is given, the current pensize is returned.

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)  # from here on lines of width 10 are drawn
```

turtle. **pen(pen=None, **pendict)**

**Parameters:**
- **pen** – a dictionary with some or all of the below listed keys
- **pendict** – one or more keyword-arguments with the below listed keys as keywords

Return or set the pen’s attributes in a “pen-dictionary” with the following key/value pairs:

- “shown”: True/False
- “pendown”: True/False
- “pencolor”: color-string or color-tuple
- "fillcolor": color-string or color-tuple
- "pensize": positive number
- "speed": number in range 0..10
- "resizemode": "auto" or "user" or "noresize"
- "stretchfactor": (positive number, positive number)
- "outline": positive number
- "tilt": number

This dictionary can be used as argument for a subsequent call to `pen()` to restore the former pen-state. Moreover one or more of these attributes can be provided as keyword-arguments. This can be used to set several pen attributes in one statement.

```python
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'), ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'), ('shown', True), ('speed', 9), ('stretchfactor', (1, 1)), ('tilt', 0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", ")
>>> turtle.penup()
>>> sorted(turtle.pen().items())
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow'), ('pendown', False), ('pensize', 10), ('resizemode', 'noresize'), ('shown', True), ('speed', 9), ('stretchfactor', (1, 1)), ('tilt', 0)]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red'), ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'), ('shown', True), ('speed', 9), ('stretchfactor', (1, 1)), ('tilt', 0)]
```

`turtle.isdown()`

Return `True` if pen is down, `False` if it’s up.

```python
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```
turtle.\texttt{pencolor}(\ast \texttt{args})

Return or set the pencolor.

Four input formats are allowed:

\texttt{pencolor()}

Return the current pencolor as color specification string or as a tuple (see example). May be used as input to another color/pencolor/fillcolor call.

\texttt{pencolor(colorstring)}

Set pencolor to \textit{colorstring}, which is a Tk color specification string, such as \texttt{"red"}, \texttt{"yellow"}, or \texttt{"#33cc8c"}.

\texttt{pencolor((r, g, b))}

Set pencolor to the RGB color represented by the tuple of \texttt{r}, \texttt{g}, and \texttt{b}. Each of \texttt{r}, \texttt{g}, and \texttt{b} must be in the range 0..colormode, where colormode is either 1.0 or 255 (see \texttt{colormode()}).

\texttt{pencolor(r, g, b)}

Set pencolor to the RGB color represented by \texttt{r}, \texttt{g}, and \texttt{b}. Each of \texttt{r}, \texttt{g}, and \texttt{b} must be in the range 0..colormode.

If turtleshape is a polygon, the outline of that polygon is drawn with the newly set pencolor.

\begin{verbatim}
>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
\end{verbatim}
turtle.fillcolor(*args)

Return or set the fillcolor.

Four input formats are allowed:

fillcolor()

Return the current fillcolor as color specification string, possibly in tuple format (see example). May be used as input to another color/pencolor/fillcolor call.

fillcolor(colorstring)

Set fillcolor to colorstring, which is a Tk color specification string, such as "red", "yellow", or "#33cc8c".

fillcolor((r, g, b))

Set fillcolor to the RGB color represented by the tuple of r, g, and b. Each of r, g, and b must be in the range 0..colormode, where colormode is either 1.0 or 255 (see colormode()).

fillcolor(r, g, b)

Set fillcolor to the RGB color represented by r, g, and b. Each of r, g, and b must be in the range 0..colormode.

If turtleshape is a polygon, the interior of that polygon is drawn with the newly set fillcolor.
turtle.fillcolor(col)
turtle.fillcolor()
(50, 193, 143)
turtle.fillcolor('#ffffff')
turtle.fillcolor()
(255, 255, 255)

turtle.color(*args)
Return or set pencolor and fillcolor.

Several input formats are allowed. They use 0 to 3 arguments as follows:

color()
Return the current pencolor and the current fillcolor as a pair of color specification strings or tuples as returned by pencolor() and fillcolor().

color(colorstring), color((r, g, b)), color(r, g, b)
Inputs as in pencolor(), set both, fillcolor and pencolor, to the given value.

color(colorstring1, colorstring2), color((r1, g1, b1), (r2, g2, b2))
Equivalent to pencolor(colorstring1) and fillcolor(colorstring2) and analogously if the other input format is used.

If turtleshape is a polygon, outline and interior of that polygon is drawn with the newly set colors.

color("red", "green")
color() ('red', 'green')
color("#285078", "#a0c8f0")
color() ((40, 80, 120), (160, 200, 240))
See also: Screen method `colormode()`.

### 24.5.3.4.3. Filling

`turtle.fill(flag)`

**Parameters:** `flag` – True/False (or 1/0 respectively)

Call `fill(True)` before drawing the shape you want to fill, and `fill(False)` when done. When used without argument: return `fillstate` (True if filling, False else).

```python
>>> turtle.fill(True)
>>> for _ in range(3):
...    turtle.forward(100)
...    turtle.left(120)
...    turtle.fill(False)
```

`turtle.begin_fill()`

Call just before drawing a shape to be filled. Equivalent to `fill(True)`.

`turtle.end_fill()`

Fill the shape drawn after the last call to `begin_fill()`. Equivalent to `fill(False)`.

```python
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

### 24.5.3.4.4. More drawing control

`turtle.reset()`

Delete the turtle’s drawings from the screen, re-center the turtle and set variables to the default values.
turtle.goto(0,-22)
turtle.left(100)
turtle.position() (0.00,-22.00)
turtle.heading() 100.0
turtle.reset()
turtle.position() (0.00,0.00)
turtle.heading() 0.0

turtle.clear()
Delete the turtle’s drawings from the screen. Do not move turtle. State and position of the turtle as well as drawings of other turtles are not affected.

turtle.write(arg, move=False, align="left", font=("Arial", 8, "normal"))

Parameters:
- **arg** – object to be written to the TurtleScreen
- **move** – True/False
- **align** – one of the strings “left”, “center” or right”
- **font** – a triple (fontname, fontsize, fonttype)

Write text - the string representation of arg - at the current turtle position according to align ("left", “center” or right") and with the given font. If move is true, the pen is moved to the bottom-right corner of the text. By default, move is False.

>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)

24.5.3.5. Turtle state

24.5.3.5.1. Visibility

turtle.hideturtle()
turtle.ht()
Make the turtle invisible. It’s a good idea to do this while you’re in the middle of doing some complex drawing, because hiding the turtle speeds up the drawing observably.

```python
>>> turtle.hideturtle()
```

turtle. `showturtle()`
turtle. `st()`
Make the turtle visible.

```python
>>> turtle.showturtle()
```

turtle. `isvisible()`
Return `True` if the Turtle is shown, `False` if it’s hidden.

```python
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

**24.5.3.5.2. Appearance**

turtle. `shape(name=None)`

**Parameters:** `name` — a string which is a valid shapename

Set turtle shape to shape with given `name` or, if name is not given, return name of current shape. Shape with `name` must exist in the TurtleScreen’s shape dictionary. Initially there are the following polygon shapes: “arrow”, “turtle”, “circle”, “square”, “triangle”, “classic”. To learn about how to deal with shapes see Screen method `register_shape()`.

```python
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
```
turtle.

---

**turtle.resizemode(rmode=None)**

Parameters:

- **rmode** – one of the strings “auto”, “user”, “noresize”

Set resizemode to one of the values: “auto”, “user”, “noresize”. If `rmode` is not given, return current resizemode. Different resizemodes have the following effects:

- “auto”: adapts the appearance of the turtle corresponding to the value of pensize.
- “user”: adapts the appearance of the turtle according to the values of stretchfactor and outlinewidth (outline), which are set by `shapesize()`.
- “noresize”: no adaption of the turtle’s appearance takes place.

`resizemode("user")` is called by `shapesize()` when used with arguments.

```python
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

---

**turtle.shapesize(stretch_wid=None, stretch_len=None, outline=None)**

**turtle.turtlesize(stretch_wid=None, stretch_len=None, outline=None)**

Parameters:

- **stretch_wid** – positive number
- **stretch_len** – positive number
- **outline** – positive number

Return or set the pen’s attributes x/y-stretchfactors and/or outline. Set resizemode to “user”. If and only if resizemode is set to “user”, the turtle will be displayed stretched according to its
stretchfactors: *stretch_wid* is stretchfactor perpendicular to its orientation, *stretch_len* is stretchfactor in direction of its orientation, *outline* determines the width of the shapes’s outline.

```python
>>> turtle.shapesize()
(1, 1, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

turtle.**tilt**(angle)

**Parameters:** angle – a number

Rotate the turtleshape by *angle* from its current tilt-angle, but do *not* change the turtle’s heading (direction of movement).

```python
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5, 2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

turtle.**settiltangle**(angle)

**Parameters:** angle – a number

Rotate the turtleshape to point in the direction specified by *angle*, regardless of its current tilt-angle. *Do not* change the turtle’s heading (direction of movement).

```python
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5, 2)
>>> turtle.settiltangle(45)
>>> turtle.fd(50)
>>> turtle.settiltangle(-45)
```
turtle.fd(50)

turtle.tiltangle()

Return the current tilt-angle, i.e. the angle between the orientation of the turtleshape and the heading of the turtle (its direction of movement).

turtle.reset()
turtle.shape("circle")
turtle.shapesize(5,2)
turtle.tilt(45)
turtle.tiltangle()

45.0

24.5.3.6. Using events

turtle.onclick(fun, btn=1, add=None)

Parameters:

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **num** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – True Or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-click events on this turtle. If *fun* is None, existing bindings are removed. Example for the anonymous turtle, i.e. the procedural way:

```python
>>> def turn(x, y):
...    left(180)
...    ...
>>> onclick(turn)  # Now clicking into the turtle will turn
>>> onclick(None)  # event-binding will be removed
```

turtle.onrelease(fun, btn=1, add=None)
Parameters:

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **num** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-button-release events on this turtle. If *fun* is None, existing bindings are removed.

```python
class MyTurtle(Turtle):
    ...
    def glow(self, x, y):
        ...
        self.fillcolor("red")
    ...
    def unglow(self, x, y):
        self.fillcolor("")
    ...

turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)  # clicking on turtle turns fillcolor red
>>> turtle.onrelease(turtle.unglow)  # releasing turns it to transparent
```

**turtle.ondrag**(*fun*, *btn=1*, *add=None*)

Parameters:

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **num** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-move events on this turtle. If *fun* is None, existing bindings are removed.

Remark: Every sequence of mouse-move-events on a turtle is preceded by a mouse-click event on that turtle.
Subsequently, clicking and dragging the Turtle will move it across the screen thereby producing handdrawings (if pen is down).

```python
>>> turtle.ondrag(turtle.goto)
```

**turtle.**`mainloop()`

**turtle.**`done()`

Starts event loop - calling Tkinter’s `mainloop` function. Must be the last statement in a turtle graphics program.

```python
>>> turtle.mainloop()
```

### 24.5.3.7. Special Turtle methods

**turtle.**`begin_poly()`

Start recording the vertices of a polygon. Current turtle position is first vertex of polygon.

**turtle.**`end_poly()`

Stop recording the vertices of a polygon. Current turtle position is last vertex of polygon. This will be connected with the first vertex.

**turtle.**`get_poly()`

Return the last recorded polygon.

```python
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

**turtle.**`clone()`

Create and return a clone of the turtle with same position,
heading and turtle properties.

```python
>>> mick = Turtle()
>>> joe = mick.clone()
```

turtle. **getturtle()**

turtle. **getpen()**

Return the Turtle object itself. Only reasonable use: as a function to return the “anonymous turtle”:

```python
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

turtle. **getscreen()**

Return the **TurtleScreen** object the turtle is drawing on. TurtleScreen methods can then be called for that object.

```python
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

turtle. **setundobuffer(size)**

**Parameters:** size – an integer or None

Set or disable undobuffer. If size is an integer an empty undobuffer of given size is installed. size gives the maximum number of turtle actions that can be undone by the `undo()` method/function. If `size` is `None`, the undobuffer is disabled.

```python
>>> turtle.setundobuffer(42)
```

turtle. **undobufferentries()**

Return number of entries in the undobuffer.
```python
>>> while undobufferentries():
    ...    undo()
```

`turtle.tracer(flag=None, delay=None)`
A replica of the corresponding TurtleScreen method.

*Deprecated since version 2.6.*

`turtle.window_width()`
`turtle.window_height()`
Both are replicas of the corresponding TurtleScreen methods.

*Deprecated since version 2.6.*

24.5.3.8. Excursus about the use of compound shapes

To use compound turtle shapes, which consist of several polygons of different color, you must use the helper class `Shape` explicitly as described below:

1. Create an empty Shape object of type “compound”.

2. Add as many components to this object as desired, using the `addcomponent()` method.

For example:

```python
>>> s = Shape("compound")
>>> poly1 = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s.addcomponent(poly1, "red", "blue")
>>> poly2 = ((0,0),(10,-5),(-10,-5))
>>> s.addcomponent(poly2, "blue", "red")
```

3. Now add the Shape to the Screen’s shapelist and use it:
```python
>>> register_shape("myshape", s)
>>> shape("myshape")
```

**Note:** The Shape class is used internally by the `register_shape()` method in different ways. The application programmer has to deal with the Shape class *only* when using compound shapes like shown above!
24.5.4. Methods of TurtleScreen/Screen and corresponding functions

Most of the examples in this section refer to a TurtleScreen instance called screen.

24.5.4.1. Window control

turtle. bgcolor(*args)

**Parameters:**

- **args** – a color string or three numbers in the range 0..colormode or a 3-tuple of such numbers

Set or return background color of the TurtleScreen.

```python
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128, 0, 128)
```

turtle. bgpic(picname=None)

**Parameters:**

- **picname** – a string, name of a gif-file or "nopic", or None

Set background image or return name of current background image. If **picname** is a filename, set the corresponding image as background. If **picname** is "nopic", delete background image, if present. If **picname** is None, return the filename of the current background image.

```python
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```
turtle. `clear()`

turtle. `clearscreen()`

Delete all drawings and all turtles from the TurtleScreen. Reset the now empty TurtleScreen to its initial state: white background, no background image, no event bindings and tracing on.

**Note:** This TurtleScreen method is available as a global function only under the name `clearscreen`. The global function `clear` is another one derived from the Turtle method `clear`.

turtle. `reset()`

turtle. `resetscreen()`

Reset all Turtles on the Screen to their initial state.

**Note:** This TurtleScreen method is available as a global function only under the name `resetscreen`. The global function `reset` is another one derived from the Turtle method `reset`.

turtle. `screensize(canvwidth=None, canvheight=None, bg=None)`

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>canvwidth</code> – positive integer, new width of canvas in pixels</td>
</tr>
<tr>
<td><code>canvheight</code> – positive integer, new height of canvas in pixels</td>
</tr>
<tr>
<td><code>bg</code> – colorstring or color-tuple, new background color</td>
</tr>
</tbody>
</table>

If no arguments are given, return current (canvaswidth, canvashheight). Else resize the canvas the turtles are drawing on. Do not alter the drawing window. To observe hidden parts of the canvas, use the scrollbars. With this method, one can make visible those parts of a drawing which were outside the canvas before.

```python
>>> screen.screensize()
```
(400, 300)
>>> screen.screensize(2000, 1500)
>>> screen.screensize()
(2000, 1500)

e.g. to search for an erroneously escaped turtle ;-)

turtle. setworldcoordinates(llx, lly, urx, ury)

Parameters:
- **llx** – a number, x-coordinate of lower left corner of canvas
- **lly** – a number, y-coordinate of lower left corner of canvas
- **urx** – a number, x-coordinate of upper right corner of canvas
- **ury** – a number, y-coordinate of upper right corner of canvas

Set up user-defined coordinate system and switch to mode “world” if necessary. This performs a screen.reset(). If mode “world” is already active, all drawings are redrawn according to the new coordinates.

**ATTENTION**: in user-defined coordinate systems angles may appear distorted.

>>> screen.reset()
>>> screen.setworldcoordinates(-50, -7.5, 50, 7.5)
>>> for _ in range(72):
...     left(10)
...     ..."
>>> for _ in range(8):
...     left(45); fd(2)  # a regular octagon

### 24.5.4.2. Animation control

turtle. delay(delay=None)

**Parameters:** delay – positive integer
Set or return the drawing delay in milliseconds. (This is approximately the time interval between two consecutive canvas updates.) The longer the drawing delay, the slower the animation.

Optional argument:

```python
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

turtle.tracer($n=None, \text{ } delay=None$)

Parameters:
- $n$ – nonnegative integer
- $delay$ – nonnegative integer

Turn turtle animation on/off and set delay for update drawings. If $n$ is given, only each $n$-th regular screen update is really performed. (Can be used to accelerate the drawing of complex graphics.) Second argument sets delay value (see `delay()`).

```python
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

turtle.update()

Perform a TurtleScreen update. To be used when tracer is turned off.

See also the RawTurtle/Turtle method `speed()`.

24.5.4.3. Using screen events

turtle.listen($xdummy=None, \text{ } ydummy=None$)

Set focus on TurtleScreen (in order to collect key-events).
Dummy arguments are provided in order to be able to pass `listen()` to the `onclick` method.

```python
turtle.onkey(fun, key)
```

**Parameters:**
- `fun` – a function with no arguments or `None`
- `key` – a string: key (e.g. “a”) or key-symbol (e.g. “space”)

Bind `fun` to key-release event of `key`. If `fun` is `None`, event bindings are removed. Remark: in order to be able to register key-events, TurtleScreen must have the focus. (See method `listen()`.)

```python
>>> def f():
...     fd(50)
...     lt(60)
... >>> screen.onkey(f, "Up")
>>> screen.listen()
```

```python
turtle.onclick(fun, btn=1, add=None)
turtle.onscreenclick(fun, btn=1, add=None)
```

**Parameters:**
- `fun` – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- `num` – number of the mouse-button, defaults to 1 (left mouse button)
- `add` – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind `fun` to mouse-click events on this screen. If `fun` is `None`, existing bindings are removed.

Example for a TurtleScreen instance named `screen` and a Turtle instance named `turtle`:

```python
>>> screen.onclick(turtle.goto) # Subsequently clicking into
```
Note: This TurtleScreen method is available as a global function only under the name `onscreenclick`. The global function `onclick` is another one derived from the Turtle method `onclick`.

```python
turtle.ontimer(fun, t=0)
```

Parameters:
- `fun` – a function with no arguments
- `t` – a number >= 0

Install a timer that calls `fun` after `t` milliseconds.

```python
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...     screen.ontimer(f, 250)
>>> f()  # makes the turtle march around
>>> running = False
```

24.5.4.4. Settings and special methods

```python
turtle.mode(mode=None)
```

Parameters:
- `mode` – one of the strings “standard”, “logo” or “world”

Set turtle mode (“standard”, “logo” or “world”) and perform reset. If mode is not given, current mode is returned.

Mode “standard” is compatible with old `turtle`. Mode “logo” is compatible with most Logo turtle graphics. Mode “world” uses user-defined “world coordinates”. **Attention:** in this mode angles appear distorted if x/y unit-ratio doesn’t equal 1.
### Mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Initial turtle heading</th>
<th>positive angles</th>
</tr>
</thead>
<tbody>
<tr>
<td>“standard”</td>
<td>to the right (east)</td>
<td>counterclockwise</td>
</tr>
<tr>
<td>“logo”</td>
<td>upward (north)</td>
<td>clockwise</td>
</tr>
</tbody>
</table>

```python
>>> mode("logo")  # resets turtle heading to north
>>> mode()
'logo'
```

#### turtle.colormode(\(cmode=\)None)

**Parameters:** \(c\)mode – one of the values 1.0 or 255

Return the colormode or set it to 1.0 or 255. Subsequently \(r, g, b\) values of color triples have to be in the range 0..\(c\)mode.

```python
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
...  
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

#### turtle.getcanvas()

Return the Canvas of this TurtleScreen. Useful for insiders who know what to do with a Tkinter Canvas.

```python
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas instance at 0x...>
```

#### turtle.getshapes()

Return a list of names of all currently available turtle shapes.

```python
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```
turtle.

**register_shape**(*name*, *shape=None*)

**addshape**(*name*, *shape=None*)

There are three different ways to call this function:

1. *name* is the name of a gif-file and *shape* is *None*: Install the corresponding image shape.

```python
>>> screen.register_shape("turtle.gif")
```

**Note:** Image shapes do not rotate when turning the turtle, so they do not display the heading of the turtle!

2. *name* is an arbitrary string and *shape* is a tuple of pairs of coordinates: Install the corresponding polygon shape.

```python
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-
```

3. *name* is an arbitrary string and *shape* is a (compound) **Shape** object: Install the corresponding compound shape.

Add a turtle shape to TurtleScreen’s shapelist. Only thusly registered shapes can be used by issuing the command `shape(shapename)`.

turtle.

**turtles**()

Return the list of turtles on the screen.

```python
>>> for turtle in screen.turtles():
...    turtle.color("red")
```

turtle.

**window_height**()

Return the height of the turtle window.

```python
>>> screen.window_height()
480
```
**turtle.** `window_width()`  
Return the width of the turtle window.

```python
>>> screen.window_width()
640
```

24.5.4.5. Methods specific to Screen, not inherited from TurtleScreen

**turtle.** `bye()`  
Shut the turtlegenetics window.

**turtle.** `exitonclick()`  
Bind `bye()` method to mouse clicks on the Screen.

If the value “using_IDLE” in the configuration dictionary is `False` (default value), also enter mainloop. Remark: If IDLE with the `-n` switch (no subprocess) is used, this value should be set to `True` in `turtle.cfg`. In this case IDLE’s own mainloop is active also for the client script.

**turtle.** `setup(width=_CFG["width"], height=_CFG["height"], startx=_CFG["leftright"], starty=_CFG["topbottom"])`  
Set the size and position of the main window. Default values of arguments are stored in the configuration dictionary and can be changed via a `turtle.cfg` file.

**Parameters:**  
- **width** – if an integer, a size in pixels, if a float, a fraction of the screen; default is 50% of screen  
- **height** – if an integer, the height in pixels, if a float, a fraction of the screen; default is 75% of screen  
- **startx** – if positive, starting position in pixels from the left edge of the screen, if negative
from the right edge, if None, center window horizontally

- **starty** – if positive, starting position in pixels from the top edge of the screen, if negative from the bottom edge, if None, center window vertically

```python
>>> screen.setup(width=200, height=200, startx=0, starty=0)
>>> # sets window to 200x200 pixels, in upper left
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>> # sets window to 75% of screen by 50% of screen
```

turtle.title(*titlestring*)

**Parameters:**

- **titlestring** – a string that is shown in the titlebar of the turtle graphics window

Set title of turtle window to *titlestring*.

```python
>>> screen.title("Welcome to the turtle zoo!")
```
24.5.5. The public classes of the module \texttt{turtle}

\begin{verbatim}
\textit{class turtle.RawTurtle(canvas)}
\textit{class turtle.RawPen(canvas)}
\end{verbatim}

\textbf{Parameters:} \texttt{canvas} – a \texttt{Tkinter.Canvas}, a \texttt{ScrolledCanvas} Or a \texttt{TurtleScreen}

Create a turtle. The turtle has all methods described above as “methods of Turtle/RawTurtle”.

\begin{verbatim}
\textit{class turtle.Turtle}
\end{verbatim}

Subclass of RawTurtle, has the same interface but draws on a default \texttt{Screen} object created automatically when needed for the first time.

\begin{verbatim}
\textit{class turtle.TurtleScreen(cv)}
\end{verbatim}

\textbf{Parameters:} \texttt{cv} – a \texttt{Tkinter.Canvas}

Provides screen oriented methods like \texttt{setbg()} etc. that are described above.

\begin{verbatim}
\textit{class turtle.Screen}
\end{verbatim}

Subclass of TurtleScreen, with four methods added.

\begin{verbatim}
\textit{class turtle.ScrolledCanvas(master)}
\end{verbatim}

\textbf{Parameters:} \texttt{master} – some Tkinter widget to contain the ScrolledCanvas, i.e. a Tkinter-canvas with scrollbars added

Used by class Screen, which thus automatically provides a ScrolledCanvas as playground for the turtles.

\begin{verbatim}
\textit{class turtle.Shape(type_, data)}
\end{verbatim}

\textbf{Parameters:} \texttt{type_} – one of the strings “polygon”, “image”, “compound”
Data structure modeling shapes. The pair (type_, data) must follow this specification:

<table>
<thead>
<tr>
<th>type_</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>“polygon”</td>
<td>a polygon-tuple, i.e. a tuple of pairs of coordinates</td>
</tr>
<tr>
<td>“image”</td>
<td>an image (in this form only used internally!)</td>
</tr>
<tr>
<td>“compound”</td>
<td>None (a compound shape has to be constructed using the addcomponent() method)</td>
</tr>
</tbody>
</table>

addcomponent(poly, fill, outline=None)

Parameters:
- poly – a polygon, i.e. a tuple of pairs of numbers
- fill – a color the poly will be filled with
- outline – a color for the poly’s outline (if given)

Example:

```python
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use register_shape
```

See Excursus about the use of compound shapes.

class turtle.Vec2D(x, y)

A two-dimensional vector class, used as a helper class for implementing turtle graphics. May be useful for turtle graphics programs too. Derived from tuple, so a vector is a tuple!

Provides (for a, b vectors, k number):

- a + b vector addition
- a - b vector subtraction
- a * b inner product
- k * a and a * k multiplication with scalar
• \texttt{abs(a)} absolute value of \texttt{a}
• \texttt{a.rotate(angle)} rotation
24.5.6. Help and configuration

24.5.6.1. How to use help

The public methods of the Screen and Turtle classes are documented extensively via docstrings. So these can be used as online-help via the Python help facilities:

- When using IDLE, tooltips show the signatures and first lines of the docstrings of typed in function-/method calls.

- Calling help() on methods or functions displays the docstrings:

```python
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
  Set or return backgroundcolor of the TurtleScreen.
  Arguments (if given): a color string or three numbers in the range 0..colormode or a 3-tuple of such numbers.

>>> screen.bgcolor("orange")
>>> screen.bgcolor()
"orange"
>>> screen.bgcolor(0.5,0,0.5)
>>> screen.bgcolor()
"#800080"

>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
  Pull the pen up -- no drawing when moving.
  Aliases: penup | pu | up
  No argument
```
The docstrings of the functions which are derived from methods have a modified form:

```python
>>> turtle.penup()
```

```python
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.
    
    Arguments (if given): a color string or three numbers in the range 0..colormode or a 3-tuple of such numbers.

    Example:::
    
    >>> bgcolor("orange")
    >>> bgcolor()
    "orange"
    >>> bgcolor(0.5,0,0.5)
    >>> bgcolor()
    "#800080"

>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

   Aliases: penup | pu | up

    No argument

    Example:
    
    >>> penup()
```

These modified docstrings are created automatically together with the function definitions that are derived from the methods at import time.

24.5.6.2. Translation of docstrings into different
There is a utility to create a dictionary the keys of which are the method names and the values of which are the docstrings of the public methods of the classes Screen and Turtle.

```python
turtle.write_docstringdict(filename="turtle_docstringdict")
```

**Parameters:** filename – a string, used as filename

Create and write docstring-dictionary to a Python script with the given filename. This function has to be called explicitly (it is not used by the turtle graphics classes). The docstring dictionary will be written to the Python script `filename.py`. It is intended to serve as a template for translation of the docstrings into different languages.

If you (or your students) want to use `turtle` with online help in your native language, you have to translate the docstrings and save the resulting file as e.g. `turtle_docstringdict_german.py`.

If you have an appropriate entry in your `turtle.cfg` file this dictionary will be read in at import time and will replace the original English docstrings.

At the time of this writing there are docstring dictionaries in German and in Italian. (Requests please to glingl@aon.at.)

### 24.5.6.3. How to configure Screen and Turtles

The built-in default configuration mimics the appearance and behaviour of the old turtle module in order to retain best possible compatibility with it.

If you want to use a different configuration which better reflects the features of this module or which better fits to your needs, e.g. for use
in a classroom, you can prepare a configuration file `turtle.cfg` which will be read at import time and modify the configuration according to its settings.

The built in configuration would correspond to the following `turtle.cfg`:

```plaintext
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

Short explanation of selected entries:

- The first four lines correspond to the arguments of the `Screen.setup()` method.
- Line 5 and 6 correspond to the arguments of the method `Screen.screensize()`.
- `shape` can be any of the built-in shapes, e.g. arrow, turtle, etc. For more info try `help(shape)`.
- If you want to use no fillcolor (i.e. make the turtle transparent), you have to write `fillcolor = ""` (but all nonempty strings must not have quotes in the cfg-file).
- If you want to reflect the turtle its state, you have to use
resizemode = auto.

- If you set e.g. `language = italian` the docstringdict `turtle_docstringdict_italian.py` will be loaded at import time (if present on the import path, e.g. in the same directory as `turtle`.

- The entries `exampleturtle` and `examplescreen` define the names of these objects as they occur in the docstrings. The transformation of method-docstrings to function-docstrings will delete these names from the docstrings.

- `using_IDLE`: Set this to `True` if you regularly work with IDLE and its `-n` switch ("no subprocess"). This will prevent `exitonclick()` to enter the mainloop.

There can be a `turtle.cfg` file in the directory where `turtle` is stored and an additional one in the current working directory. The latter will override the settings of the first one.

The `Demo/turtle` directory contains a `turtle.cfg` file. You can study it as an example and see its effects when running the demos (preferably not from within the demo-viewer).
24.5.7. Demo scripts

There is a set of demo scripts in the turtledemo directory located in the Demo/turtle directory in the source distribution.

It contains:

- a set of 15 demo scripts demonstrating different features of the new module turtle
- a demo viewer turtleDemo.py which can be used to view the source code of the scripts and run them at the same time. 14 of the examples can be accessed via the Examples menu; all of them can also be run standalone.
- The example turtledemo_two_canvases.py demonstrates the simultaneous use of two canvases with the turtle module. Therefore it only can be run standalone.
- There is a turtle.cfg file in this directory, which also serves as an example for how to write and use such files.

The demoscripts are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytedesign</td>
<td>complex classical turtle graphics pattern</td>
<td>tracer(), delay, update()</td>
</tr>
<tr>
<td>chaos</td>
<td>graphs Verhulst dynamics, shows that computer’s computations can generate results sometimes against the common sense expectations</td>
<td>world coordinates</td>
</tr>
<tr>
<td>clock</td>
<td>analog clock showing time of your computer</td>
<td>turtles as clock’s hands, ontimer</td>
</tr>
<tr>
<td>colormixer</td>
<td>experiment with r, g, b</td>
<td>ondrag()</td>
</tr>
<tr>
<td>fractalcurves</td>
<td>Hilbert &amp; Koch curves</td>
<td>recursion</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>lindenmayer</td>
<td>ethnomathematics (indian kolams)</td>
<td>L-System</td>
</tr>
<tr>
<td>minimal_hanoi</td>
<td>Towers of Hanoi</td>
<td>Rectangular Turtles as Hanoi discs (shape, shapesize)</td>
</tr>
<tr>
<td>paint</td>
<td>super minimalistic drawing program</td>
<td>onclick()</td>
</tr>
<tr>
<td>peace</td>
<td>elementary</td>
<td>turtle: appearance and animation</td>
</tr>
<tr>
<td>penrose</td>
<td>aperiodic tiling with kites and darts</td>
<td>stamp()</td>
</tr>
<tr>
<td>planet_and_moon</td>
<td>simulation of gravitational system</td>
<td>compound shapes, Vec2D</td>
</tr>
<tr>
<td>tree</td>
<td>a (graphical) breadth first tree (using generators)</td>
<td>clone()</td>
</tr>
<tr>
<td>wikipedia</td>
<td>a pattern from the wikipedia article on turtle graphics</td>
<td>clone(), undo()</td>
</tr>
<tr>
<td>yingyang</td>
<td>another elementary example</td>
<td>circle()</td>
</tr>
</tbody>
</table>

Have fun!
24.6. IDLE

IDLE is Python’s Integrated Development and Learning Environment.

IDLE has the following features:

- coded in 100% pure Python, using the `tkinter` GUI toolkit
- cross-platform: works mostly the same on Windows, Unix, and Mac OS X
- Python shell window (interactive interpreter) with colorizing of code input, output, and error messages
- multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features
- search within any window, replace within editor windows, and search through multiple files (grep)
- debugger with persistent breakpoints, stepping, and viewing of global and local namespaces
- configuration, browsers, and other dialogs
24.6.1. Menus

IDLE has two main window types, the Shell window and the Editor window. It is possible to have multiple editor windows simultaneously. Output windows, such as used for Edit / Find in Files, are a subtype of edit window. They currently have the same top menu as Editor windows but a different default title and context menu.

IDLE's menus dynamically change based on which window is currently selected. Each menu documented below indicates which window type it is associated with.

24.6.1.1. File menu (Shell and Editor)

New File
    Create a new file editing window.

Open...
    Open an existing file with an Open dialog.

Recent Files
    Open a list of recent files. Click one to open it.

Open Module...
    Open an existing module (searches sys.path).

Class Browser
    Show functions, classes, and methods in the current Editor file in a tree structure. In the shell, open a module first.

Path Browser
    Show sys.path directories, modules, functions, classes and methods in a tree structure.

Save
Save the current window to the associated file, if there is one. Windows that have been changed since being opened or last saved have a * before and after the window title. If there is no associated file, do Save As instead.

**Save As...**
Save the current window with a Save As dialog. The file saved becomes the new associated file for the window.

**Save Copy As...**
Save the current window to different file without changing the associated file.

**Print Window**
Print the current window to the default printer.

**Close**
Close the current window (ask to save if unsaved).

**Exit**
Close all windows and quit IDLE (ask to save unsaved windows).

### 24.6.1.2. Edit menu (Shell and Editor)

**Undo**
Undo the last change to the current window. A maximum of 1000 changes may be undone.

**Redo**
Redo the last undone change to the current window.

**Cut**
Copy selection into the system-wide clipboard; then delete the selection.

**Copy**
Copy selection into the system-wide clipboard.

**Paste**
Insert contents of the system-wide clipboard into the current
The clipboard functions are also available in context menus.

Select All
Select the entire contents of the current window.

Find...
Open a search dialog with many options

Find Again
Repeat the last search, if there is one.

Find Selection
Search for the currently selected string, if there is one.

Find in Files...
Open a file search dialog. Put results in a new output window.

Replace...
Open a search-and-replace dialog.

Go to Line
Move cursor to the line number requested and make that line visible.

Show Completions
Open a scrollable list allowing selection of keywords and attributes. See Completions in the Tips sections below.

Expand Word
Expand a prefix you have typed to match a full word in the same window; repeat to get a different expansion.

Show call tip
After an unclosed parenthesis for a function, open a small window with function parameter hints.

Show surrounding parens
Highlight the surrounding parenthesis.
24.6.1.3. Format menu (Editor window only)

Indent Region
Shift selected lines right by the indent width (default 4 spaces).

Dedent Region
Shift selected lines left by the indent width (default 4 spaces).

Comment Out Region
Insert ## in front of selected lines.

Uncomment Region
Remove leading # or ## from selected lines.

Tabify Region
Turn leading stretches of spaces into tabs. (Note: We recommend using 4 space blocks to indent Python code.)

Untabify Region
Turn all tabs into the correct number of spaces.

Toggle Tabs
Open a dialog to switch between indenting with spaces and tabs.

New Indent Width
Open a dialog to change indent width. The accepted default by the Python community is 4 spaces.

Format Paragraph
Reformat the current blank-line-delimited paragraph in comment block or multiline string or selected line in a string. All lines in the paragraph will be formatted to less than N columns, where N defaults to 72.

Strip trailing whitespace
Remove any space characters after the last non-space character of a line.

24.6.1.4. Run menu (Editor window only)
Python Shell
Open or wake up the Python Shell window.

Check Module
Check the syntax of the module currently open in the Editor window. If the module has not been saved IDLE will either prompt the user to save or autosave, as selected in the General tab of the Idle Settings dialog. If there is a syntax error, the approximate location is indicated in the Editor window.

Run Module
Do Check Module (above). If no error, restart the shell to clean the environment, then execute the module. Output is displayed in the Shell window. Note that output requires use of `print` or `write`. When execution is complete, the Shell retains focus and displays a prompt. At this point, one may interactively explore the result of execution. This is similar to executing a file with `python -i file` at a command line.

24.6.1.5. Shell menu (Shell window only)

View Last Restart
Scroll the shell window to the last Shell restart.

Restart Shell
Restart the shell to clean the environment.

Interrupt Execution
Stop a running program.

24.6.1.6. Debug menu (Shell window only)

Go to File/Line
Look on the current line, with the cursor, and the line above for a filename and line number. If found, open the file if not already open, and show the line. Use this to view source lines referenced
in an exception traceback and lines found by Find in Files. Also available in the context menu of the Shell window and Output windows.

Debugger (toggle)
When actived, code entered in the Shell or run from an Editor will run under the debugger. In the Editor, breakpoints can be set with the context menu. This feature is still incomplete and somewhat experimental.

Stack Viewer
Show the stack traceback of the last exception in a tree widget, with access to locals and globals.

Auto-open Stack Viewer
Toggle automatically opening the stack viewer on an unhandled exception.

24.6.1.7. Options menu (Shell and Editor)

Configure IDLE
Open a configuration dialog and change preferences for the following: fonts, indentation, keybindings, text color themes, startup windows and size, additional help sources, and extensions (see below). On OS X, open the configuration dialog by selecting Preferences in the application menu. To use a new built-in color theme (IDLE Dark) with older IDLEs, save it as a new custom theme.

Non-default user settings are saved in a .idlerc directory in the user's home directory. Problems caused by bad user configuration files are solved by editing or deleting one or more of the files in .idlerc.

Code Context (toggle)(Editor Window only)
Open a pane at the top of the edit window which shows the block
context of the code which has scrolled above the top of the window.

### 24.6.1.8. Window menu (Shell and Editor)

**Zoom Height**

Toggles the window between normal size and maximum height. The initial size defaults to 40 lines by 80 chars unless changed on the General tab of the Configure IDLE dialog.

The rest of this menu lists the names of all open windows; select one to bring it to the foreground (deiconifying it if necessary).

### 24.6.1.9. Help menu (Shell and Editor)

**About IDLE**

Display version, copyright, license, credits, and more.

**IDLE Help**

Display a help file for IDLE detailing the menu options, basic editing and navigation, and other tips.

**Python Docs**

Access local Python documentation, if installed, or start a web browser and open docs.python.org showing the latest Python documentation.

**Turtle Demo**

Run the turtledemo module with example python code and turtle drawings.

Additional help sources may be added here with the Configure IDLE dialog under the General tab.

### 24.6.1.10. Context Menus
Open a context menu by right-clicking in a window (Control-click on OS X). Context menus have the standard clipboard functions also on the Edit menu.

Cut
   Copy selection into the system-wide clipboard; then delete the selection.

Copy
   Copy selection into the system-wide clipboard.

Paste
   Insert contents of the system-wide clipboard into the current window.

Editor windows also have breakpoint functions. Lines with a breakpoint set are specially marked. Breakpoints only have an effect when running under the debugger. Breakpoints for a file are saved in the user’s .idlerc directory.

Set Breakpoint
   Set a breakpoint on the current line.

Clear Breakpoint
   Clear the breakpoint on that line.

Shell and Output windows have the following.

Go to file/line
   Same as in Debug menu.
24.6.2. Editing and navigation

In this section, ‘C’ refers to the Control key on Windows and Unix and the Command key on Mac OSX.

- **Backspace** deletes to the left; **Del** deletes to the right
- **C-Backspace** delete word left; **C-Del** delete word to the right
- **Arrow keys and Page Up/Page Down** to move around
- **C-LeftArrow** and **C-RightArrow** moves by words
- **Home/End** go to begin/end of line
- **C-Home/C-End** go to begin/end of file

- Some useful Emacs bindings are inherited from Tcl/Tk:
  
  - **C-a** beginning of line
  - **C-e** end of line
  - **C-k** kill line (but doesn’t put it in clipboard)
  - **C-l** center window around the insertion point
  - **C-b** go backwards one character without deleting (usually you can also use the cursor key for this)
  - **C-f** go forward one character without deleting (usually you can also use the cursor key for this)
  - **C-p** go up one line (usually you can also use the cursor key for this)
  - **C-d** delete next character

Standard keybindings (like **c-c** to copy and **c-v** to paste) may work. Keybindings are selected in the Configure IDLE dialog.
24.6.2.1. Automatic indentation

After a block-opening statement, the next line is indented by 4 spaces (in the Python Shell window by one tab). After certain keywords (break, return etc.) the next line is dedented. In leading indentation, Backspace deletes up to 4 spaces if they are there. Tab inserts spaces (in the Python Shell window one tab), number depends on Indent width. Currently tabs are restricted to four spaces due to Tcl/Tk limitations.

See also the indent/dedent region commands in the edit menu.

24.6.2.2. Completions

Completions are supplied for functions, classes, and attributes of classes, both built-in and user-defined. Completions are also provided for filenames.

The AutoCompleteWindow (ACW) will open after a predefined delay (default is two seconds) after a ‘.’ or (in a string) an os.sep is typed. If after one of those characters (plus zero or more other characters) a tab is typed the ACW will open immediately if a possible continuation is found.

If there is only one possible completion for the characters entered, a Tab will supply that completion without opening the ACW.

‘Show Completions’ will force open a completions window, by default the C-space will open a completions window. In an empty string, this will contain the files in the current directory. On a blank line, it will contain the built-in and user-defined functions and classes in the current name spaces, plus any modules imported. If some characters have been entered, the ACW will attempt to be more specific.
If a string of characters is typed, the ACW selection will jump to the entry most closely matching those characters. Entering a tab will cause the longest non-ambiguous match to be entered in the Editor window or Shell. Two tab in a row will supply the current ACW selection, as will return or a double click. Cursor keys, Page Up/Down, mouse selection, and the scroll wheel all operate on the ACW.

“Hidden” attributes can be accessed by typing the beginning of hidden name after a '.', e.g. '_'. This allows access to modules with __all__ set, or to class-private attributes.

Completions and the ‘Expand Word’ facility can save a lot of typing!

Completions are currently limited to those in the namespaces. Names in an Editor window which are not via __main__ and sys.modules will not be found. Run the module once with your imports to correct this situation. Note that IDLE itself places quite a few modules in sys.modules, so much can be found by default, e.g. the re module.

If you don’t like the ACW popping up unbidden, simply make the delay longer or disable the extension.

24.6.2.3. Calltips

A calltip is shown when one types ( after the name of an accessible function. A name expression may include dots and subscripts. A calltip remains until it is clicked, the cursor is moved out of the argument area, or ) is typed. When the cursor is in the argument part of a definition, the menu or shortcut display a calltip.

A calltip consists of the function signature and the first line of the docstring. For builtins without an accessible signature, the calltip
consists of all lines up the fifth line or the first blank line. These
details may change.

The set of accessible functions depends on what modules have
been imported into the user process, including those imported by
Idle itself, and what definitions have been run, all since the last
restart.

For example, restart the Shell and enter `iter tools.count()`. A calltip
appears because Idle imports itertools into the user process for its
own use. (This could change.) Enter `turtle.write()` and nothing
appears. Idle does not import turtle. The menu or shortcut do nothing
either. Enter `import turtle` and then `turtle.write()` will work.

In an editor, import statements have no effect until one runs the file.
One might want to run a file after writing the import statements at the
top, or immediately run an existing file before editing.

24.6.2.4. Python Shell window

- `c-c` interrupts executing command
- `c-d` sends end-of-file; closes window if typed at a `>>>` prompt
- `Alt-/` (Expand word) is also useful to reduce typing

Command history

- `Alt-p` retrieves previous command matching what you have
typed. On OS X use `c-p`.
- `Alt-n` retrieves next. On OS X use `c-n`.
- Return while on any previous command retrieves that command
24.6.2.5. Text colors

Idle defaults to black on white text, but colors text with special meanings. For the shell, these are shell output, shell error, user output, and user error. For Python code, at the shell prompt or in an editor, these are keywords, builtin class and function names, names following `class` and `def`, strings, and comments. For any text window, these are the cursor (when present), found text (when possible), and selected text.

Text coloring is done in the background, so uncolorized text is occasionally visible. To change the color scheme, use the Configure IDLE dialog Highlighting tab. The marking of debugger breakpoint lines in the editor and text in popups and dialogs is not user-configurable.
24.6.3. Startup and code execution

Upon startup with the \texttt{-s} option, IDLE will execute the file referenced by the environment variables \texttt{IDLESTARTUP} or \texttt{PYTHONSTARTUP}. IDLE first checks for \texttt{IDLESTARTUP}; if \texttt{IDLESTARTUP} is present the file referenced is run. If \texttt{IDLESTARTUP} is not present, IDLE checks for \texttt{PYTHONSTARTUP}. Files referenced by these environment variables are convenient places to store functions that are used frequently from the IDLE shell, or for executing import statements to import common modules.

In addition, \texttt{Tk} also loads a startup file if it is present. Note that the Tk file is loaded unconditionally. This additional file is \texttt{.Idle.py} and is looked for in the user's home directory. Statements in this file will be executed in the Tk namespace, so this file is not useful for importing functions to be used from IDLE's Python shell.

24.6.3.1. Command line usage

```
```

- \texttt{-c command} run command in the shell window
- \texttt{-d} enable debugger and open shell window
- \texttt{-e} open editor window
- \texttt{-h} print help message with legal combinations and exit
- \texttt{-i} open shell window
- \texttt{-r file} run file in shell window
- \texttt{-s} run $IDLESTARTUP or $PYTHONSTARTUP first, in shell
- \texttt{-t title} set title of shell window
- \texttt{-} run stdin in shell (- must be last option before args)

If there are arguments:

- If \texttt{-}, \texttt{-c}, or \texttt{r} is used, all arguments are placed in
sys.argv[1:] and sys.argv[0] is set to '', '-c', or '-r'. No editor window is opened, even if that is the default set in the Options dialog.

- Otherwise, arguments are files opened for editing and sys.argv reflects the arguments passed to IDLE itself.

### 24.6.3.2. IDLE-console differences

As much as possible, the result of executing Python code with IDLE is the same as executing the same code in a console window. However, the different interface and operation occasionally affects visible results. For instance, `sys.modules` starts with more entries.

IDLE also replaces `sys.stdin`, `sys.stdout`, and `sys.stderr` with objects that get input from and send output to the Shell window. When this window has the focus, it controls the keyboard and screen. This is normally transparent, but functions that directly access the keyboard and screen will not work. If `sys` is reset with `reload(sys)`, IDLE's changes are lost and things like `input`, `raw_input`, and `print` will not work correctly.

With IDLE's Shell, one enters, edits, and recalls complete statements. Some consoles only work with a single physical line at a time. IDLE uses `exec` to run each statement. As a result, '__builtins__' is always defined for each statement.

### 24.6.3.3. Running without a subprocess

By default, IDLE executes user code in a separate subprocess via a socket, which uses the internal loopback interface. This connection is not externally visible and no data is sent to or received from the Internet. If firewall software complains anyway, you can ignore it.

If the attempt to make the socket connection fails, Idle will notify you.
Such failures are sometimes transient, but if persistent, the problem may be either a firewall blocking the connection or misconfiguration of a particular system. Until the problem is fixed, one can run Idle with the -n command line switch.

If IDLE is started with the -n command line switch it will run in a single process and will not create the subprocess which runs the RPC Python execution server. This can be useful if Python cannot create the subprocess or the RPC socket interface on your platform. However, in this mode user code is not isolated from IDLE itself. Also, the environment is not restarted when Run/Run Module (F5) is selected. If your code has been modified, you must reload() the affected modules and re-import any specific items (e.g. from foo import baz) if the changes are to take effect. For these reasons, it is preferable to run IDLE with the default subprocess if at all possible.

*Deprecated since version 3.4.*
24.6.4. Help and preferences

24.6.4.1. Additional help sources

IDLE includes a help menu entry called “Python Docs” that will open the extensive sources of help, including tutorials, available at docs.python.org. Selected URLs can be added or removed from the help menu at any time using the Configure IDLE dialog. See the IDLE help option in the help menu of IDLE for more information.

24.6.4.2. Setting preferences

The font preferences, highlighting, keys, and general preferences can be changed via Configure IDLE on the Option menu. Keys can be user defined; IDLE ships with four built in key sets. In addition a user can create a custom key set in the Configure IDLE dialog under the keys tab.

24.6.4.3. Extensions

IDLE contains an extension facility. Preferences for extensions can be changed with Configure Extensions. See the beginning of config-extensions.def in the idlelib directory for further information. The default extensions are currently:

- FormatParagraph
- AutoExpand
- ZoomHeight
- ScriptBinding
- CallTips
- ParenMatch
- AutoComplete
- CodeContext
• RstripExtension
24.7. Other Graphical User Interface Packages

Major cross-platform (Windows, Mac OS X, Unix-like) GUI toolkits are available for Python:

See also:

**PyGTK**

is a set of bindings for the GTK widget set. It provides an object oriented interface that is slightly higher level than the C one. It comes with many more widgets than Tkinter provides, and has good Python-specific reference documentation. There are also bindings to GNOME. An online tutorial is available.

**PyQt**

PyQt is a sip-wrapped binding to the Qt toolkit. Qt is an extensive C++ GUI application development framework that is available for Unix, Windows and Mac OS X. sip is a tool for generating bindings for C++ libraries as Python classes, and is specifically designed for Python. The PyQt3 bindings have a book, GUI Programming with Python: QT Edition by Boudewijn Rempt. The PyQt4 bindings also have a book, Rapid GUI Programming with Python and Qt, by Mark Summerfield.

**wxPython**

wxPython is a cross-platform GUI toolkit for Python that is built around the popular wxWidgets (formerly wxWindows) C++ toolkit. It provides a native look and feel for applications on Windows, Mac OS X, and Unix systems by using each platform’s native widgets where ever possible, (GTK+ on Unix-like systems). In addition to an extensive set of widgets, wxPython provides classes for online documentation and context sensitive help, printing, HTML viewing, low-level device
context drawing, drag and drop, system clipboard access, an XML-based resource format and more, including an ever growing library of user-contributed modules. wxPython has a book, *wxPython in Action*, by Noel Rappin and Robin Dunn.

PyGTK, PyQt, and wxPython, all have a modern look and feel and more widgets than Tkinter. In addition, there are many other GUI toolkits for Python, both cross-platform, and platform-specific. See the GUI Programming page in the Python Wiki for a much more complete list, and also for links to documents where the different GUI toolkits are compared.
25. Development Tools

The modules described in this chapter help you write software. For example, the `pydoc` module takes a module and generates documentation based on the module’s contents. The `doctest` and `unittest` modules contains frameworks for writing unit tests that automatically exercise code and verify that the expected output is produced. `2to3` can translate Python 2.x source code into valid Python 3.x code.

The list of modules described in this chapter is:

- **25.1. pydoc** — Documentation generator and online help system
- **25.2. doctest** — Test interactive Python examples
  - **25.2.1. Simple Usage: Checking Examples in Docstrings**
  - **25.2.2. Simple Usage: Checking Examples in a Text File**
  - **25.2.3. How It Works**
    - **25.2.3.1. Which Docstrings Are Examined?**
    - **25.2.3.2. How are Docstring Examples Recognized?**
    - **25.2.3.3. What’s the Execution Context?**
    - **25.2.3.4. What About Exceptions?**
    - **25.2.3.5. Option Flags**
    - **25.2.3.6. Directives**
    - **25.2.3.7. Warnings**
  - **25.2.4. Basic API**
  - **25.2.5. Unittest API**
  - **25.2.6. Advanced API**
    - **25.2.6.1. DocTest Objects**
    - **25.2.6.2. Example Objects**
    - **25.2.6.3. DocTestFinder objects**
    - **25.2.6.4. DocTestParser objects**
    - **25.2.6.5. DocTestRunner objects**
    - **25.2.6.6. OutputChecker objects**
- 25.2.7. Debugging
- 25.2.8. Soapbox
- 25.3. **unittest** — Unit testing framework
  - 25.3.1. Basic example
  - 25.3.2. Command-Line Interface
    - 25.3.2.1. Command-line options
  - 25.3.3. Test Discovery
  - 25.3.4. Organizing test code
  - 25.3.5. Re-using old test code
  - 25.3.6. Skipping tests and expected failures
  - 25.3.7. Classes and functions
    - 25.3.7.1. Test cases
      - 25.3.7.1.1. Deprecated aliases
    - 25.3.7.2. Grouping tests
    - 25.3.7.3. Loading and running tests
      - 25.3.7.3.1. load_tests Protocol
  - 25.3.8. Class and Module Fixtures
    - 25.3.8.1. setUpClass and tearDownClass
    - 25.3.8.2. setUpModule and tearDownModule
  - 25.3.9. Signal Handling
- 25.4. **2to3** - Automated Python 2 to 3 code translation
  - 25.4.1. Using 2to3
  - 25.4.2. Fixers
  - 25.4.3. **lib2to3** - 2to3's library
- 25.5. **test** — Regression tests package for Python
  - 25.5.1. Writing Unit Tests for the test package
  - 25.5.2. Running tests using the command-line interface
- 25.6. **test.test_support** — Utility functions for tests
25.1. **pydoc** — Documentation generator and online help system

*New in version 2.1.*

**Source code:** Lib/pydoc.py

The **pydoc** module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a Web browser, or saved to HTML files.

For modules, classes, functions and methods, the displayed documentation is derived from the docstring (i.e. the `__doc__` attribute) of the object, and recursively of its documentable members. If there is no docstring, **pydoc** tries to obtain a description from the block of comment lines just above the definition of the class, function or method in the source file, or at the top of the module (see `inspect.getcomments()`).

The built-in function `help()` invokes the online help system in the interactive interpreter, which uses **pydoc** to generate its documentation as text on the console. The same text documentation can also be viewed from outside the Python interpreter by running **pydoc** as a script at the operating system’s command prompt. For example, running

```
pydoc sys
```

at a shell prompt will display documentation on the **sys** module, in a style similar to the manual pages shown by the Unix **man** command. The argument to **pydoc** can be the name of a function, module, or
package, or a dotted reference to a class, method, or function within a module or module in a package. If the argument to **pydoc** looks like a path (that is, it contains the path separator for your operating system, such as a slash in Unix), and refers to an existing Python source file, then documentation is produced for that file.

| Note: | In order to find objects and their documentation, **pydoc** imports the module(s) to be documented. Therefore, any code on module level will be executed on that occasion. Use an `if __name__ == '__main__':` guard to only execute code when a file is invoked as a script and not just imported. |

When printing output to the console, **pydoc** attempts to paginate the output for easier reading. If the **PAGER** environment variable is set, **pydoc** will use its value as a pagination program.

Specifying a `-w` flag before the argument will cause HTML documentation to be written out to a file in the current directory, instead of displaying text on the console.

Specifying a `-k` flag before the argument will search the synopsis lines of all available modules for the keyword given as the argument, again in a manner similar to the Unix **man** command. The synopsis line of a module is the first line of its documentation string.

You can also use **pydoc** to start an HTTP server on the local machine that will serve documentation to visiting Web browsers. **pydoc -p 1234** will start a HTTP server on port 1234, allowing you to browse the documentation at [http://localhost:1234/](http://localhost:1234/) in your preferred Web browser. **pydoc -g** will start the server and additionally bring up a small **Tkinter**-based graphical interface to help you search for documentation pages.

When **pydoc** generates documentation, it uses the current
environment and path to locate modules. Thus, invoking `pydoc spam` documents precisely the version of the module you would get if you started the Python interpreter and typed `import spam`.

Module docs for core modules are assumed to reside in `https://docs.python.org/library/`. This can be overridden by setting the `PYTHONDOCS` environment variable to a different URL or to a local directory containing the Library Reference Manual pages.
25.2. doctest — Test interactive Python examples

The doctest module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. There are several common ways to use doctest:

- To check that a module’s docstrings are up-to-date by verifying that all interactive examples still work as documented.
- To perform regression testing by verifying that interactive examples from a test file or a test object work as expected.
- To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of “literate testing” or “executable documentation”.

Here’s a complete but small example module:

```python
""
This is the "example" module.

The example module supplies one function, factorial(). For example,

```python
>>> factorial(5)
120
""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.
    If the result is small enough to fit in an int, return an int.
    Else return a long.
    """
    [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> [factorial(long(n)) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
```
>>> factorial(30)
265252859812191058636308480000000L
>>> factorial(30L)
265252859812191058636308480000000L
>>> factorial(-1)
Traceback (most recent call last):
...
ValueError: n must be >= 0

Factorials of floats are OK, but the float must be an exact integer:
>>> factorial(30.1)
Traceback (most recent call last):
...
ValueError: n must be exact integer
>>> factorial(30.0)
265252859812191058636308480000000L

It must also not be ridiculously large:
>>> factorial(1e100)
Traceback (most recent call last):
...
OverflowError: n too large

import math
if not n >= 0:
    raise ValueError("n must be >= 0")
if math.floor(n) != n:
    raise ValueError("n must be exact integer")
if n+1 == n:  # catch a value like 1e300
    raise OverflowError("n too large")
result = 1
factor = 2
while factor <= n:
    result *= factor
    factor += 1
return result

if __name__ == "__main__":
    import doctest
doctest.testmod()

If you run example.py directly from the command line, doctest works its magic:
There’s no output! That’s normal, and it means all the examples worked. Pass `-v` to the script, and `doctest` prints a detailed log of what it’s trying, and prints a summary at the end:

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
Trying:
    [factorial(long(n)) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

And so on, eventually ending with:

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items passed all tests:
    1 tests in __main__
    8 tests in __main__.factorial
9 tests in 2 items.
9 passed and 0 failed.
Test passed.
```

That’s all you need to know to start making productive use of
doctest! Jump in. The following sections provide full details. Note that there are many examples of doctests in the standard Python test suite and libraries. Especially useful examples can be found in the standard test file `Lib/test/test_doctest.py`.
25.2.1. Simple Usage: Checking Examples in Docstrings

The simplest way to start using doctest (but not necessarily the way you'll continue to do it) is to end each module \texttt{M} with:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

\texttt{doctest} then examines docstrings in module \texttt{M}.

Running the module as a script causes the examples in the docstrings to get executed and verified:

```
python M.py
```

This won't display anything unless an example fails, in which case the failing example(s) and the cause(s) of the failure(s) are printed to stdout, and the final line of output is 
***Test Failed*** \texttt{N failures.}, where \texttt{N} is the number of examples that failed.

Run it with the \texttt{-v} switch instead:

```
python M.py -v
```

and a detailed report of all examples tried is printed to standard output, along with assorted summaries at the end.

You can force verbose mode by passing \texttt{verbose=True} to \texttt{testmod()}, or prohibit it by passing \texttt{verbose=False}. In either of those cases, \texttt{sys.argv} is not examined by \texttt{testmod()} (so passing \texttt{-v} or not has no effect).
Since Python 2.6, there is also a command line shortcut for running `testmod()`. You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the module name(s) on the command line:

```
python -m doctest -v example.py
```

This will import `example.py` as a standalone module and run `testmod()` on it. Note that this may not work correctly if the file is part of a package and imports other submodules from that package.

For more information on `testmod()`, see section Basic API.
25.2.2. Simple Usage: Checking Examples in a Text File

Another simple application of doctest is testing interactive examples in a text file. This can be done with the `testfile()` function:

```python
import doctest
doctest.testfile("example.txt")
```

That short script executes and verifies any interactive Python examples contained in the file `example.txt`. The file content is treated as if it were a single giant docstring; the file doesn't need to contain a Python program! For example, perhaps `example.txt` contains this:

```
The ``example`` module
======================

Using ``factorial``
-----------------

This is an example text file in reStructuredText format. First ```factorial``` from the ```example``` module:

```python
>>> from example import factorial
```

Now use it:

```python
>>> factorial(6)
120
```

Running `doctest.testfile("example.txt")` then finds the error in this documentation:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
```
As with `testmod()`, `testfile()` won’t display anything unless an example fails. If an example does fail, then the failing example(s) and the cause(s) of the failure(s) are printed to stdout, using the same format as `testmod()`.

By default, `testfile()` looks for files in the calling module’s directory. See section Basic API for a description of the optional arguments that can be used to tell it to look for files in other locations.

Like `testmod()`, `testfile()`'s verbosity can be set with the `-v` command-line switch or with the optional keyword argument `verbose`.

Since Python 2.6, there is also a command line shortcut for running `testfile()`. You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the file name(s) on the command line:

```
python -m doctest -v example.txt
```

Because the file name does not end with `.py`, `doctest` infers that it must be run with `testfile()`, not `testmod()`.

For more information on `testfile()`, see section Basic API.
25.2.3. How It Works

This section examines in detail how doctest works: which docstrings it looks at, how it finds interactive examples, what execution context it uses, how it handles exceptions, and how option flags can be used to control its behavior. This is the information that you need to know to write doctest examples; for information about actually running doctest on these examples, see the following sections.

25.2.3.1. Which Docstrings Are Examined?

The module docstring, and all function, class and method docstrings are searched. Objects imported into the module are not searched.

In addition, if `M.__test__` exists and “is true”, it must be a dict, and each entry maps a (string) name to a function object, class object, or string. Function and class object docstrings found from `M.__test__` are searched, and strings are treated as if they were docstrings. In output, a key \( K \) in `M.__test__` appears with name

```
<name of M>.__test__.K
```

Any classes found are recursively searched similarly, to test docstrings in their contained methods and nested classes.

*Changed in version 2.4:* A “private name” concept is deprecated and no longer documented.

25.2.3.2. How are Docstring Examples Recognized?

In most cases a copy-and-paste of an interactive console session works fine, but doctest isn’t trying to do an exact emulation of any
specific Python shell.

```python
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print "yes"
... else:
...     print "no"
...     print "NO"
...     print "NO!!"
... no
NO NO!!!
```

Any expected output must immediately follow the final `>>>` or `...` line containing the code, and the expected output (if any) extends to the next `>>>` or all-whitespace line.

The fine print:

- Expected output cannot contain an all-whitespace line, since such a line is taken to signal the end of expected output. If expected output does contain a blank line, put `<BLANKLINE>` in your doctest example each place a blank line is expected.

  New in version 2.4: `<BLANKLINE>` was added; there was no way to use expected output containing empty lines in previous versions.

- All hard tab characters are expanded to spaces, using 8-column tab stops. Tabs in output generated by the tested code are not modified. Because any hard tabs in the sample output are expanded, this means that if the code output includes hard tabs, the only way the doctest can pass is if the `NORMALIZE_WHITESPACE`
option or directive is in effect. Alternatively, the test can be rewritten to capture the output and compare it to an expected value as part of the test. This handling of tabs in the source was arrived at through trial and error, and has proven to be the least error prone way of handling them. It is possible to use a different algorithm for handling tabs by writing a custom DocTestParser class.

*Changed in version 2.4:* Expanding tabs to spaces is new; previous versions tried to preserve hard tabs, with confusing results.

- Output to stdout is captured, but not output to stderr (exception tracebacks are captured via a different means).

- If you continue a line via backslashing in an interactive session, or for any other reason use a backslash, you should use a raw docstring, which will preserve your backslashes exactly as you type them:

```python
>>> def f(x):
...   r'''Backslashes in a raw docstring: m\n'''
>>> print f.__doc__
```

Backslashes in a raw docstring: m

Otherwise, the backslash will be interpreted as part of the string. For example, the `\n` above would be interpreted as a newline character. Alternatively, you can double each backslash in the doctest version (and not use a raw string):

```python
>>> def f(x):
...   '''Backslashes in a raw docstring: m\n'''
>>> print f.__doc__
```

Backslashes in a raw docstring: m

- The starting column doesn’t matter:
and as many leading whitespace characters are stripped from the expected output as appeared in the initial '>>> ' line that started the example.

25.2.3.3. What's the Execution Context?

By default, each time doctest finds a docstring to test, it uses a shallow copy of M's globals, so that running tests doesn't change the module’s real globals, and so that one test in M can't leave behind crumbs that accidentally allow another test to work. This means examples can freely use any names defined at top-level in M, and names defined earlier in the docstring being run. Examples cannot see names defined in other docstrings.

You can force use of your own dict as the execution context by passing globs=your_dict to testmod() or testfile() instead.

25.2.3.4. What About Exceptions?

No problem, provided that the traceback is the only output produced by the example: just paste in the traceback. [1] Since tracebacks contain details that are likely to change rapidly (for example, exact file paths and line numbers), this is one case where doctest works hard to be flexible in what it accepts.

Simple example:
That doctest succeeds if `ValueError` is raised, with the `list.remove(x): x not in list` detail as shown.

The expected output for an exception must start with a traceback header, which may be either of the following two lines, indented the same as the first line of the example:

```
Traceback (most recent call last):
Traceback (innermost last):
```

The traceback header is followed by an optional traceback stack, whose contents are ignored by doctest. The traceback stack is typically omitted, or copied verbatim from an interactive session.

The traceback stack is followed by the most interesting part: the line(s) containing the exception type and detail. This is usually the last line of a traceback, but can extend across multiple lines if the exception has a multi-line detail:

```
>>> raise ValueError('multi
   line
detail')
Traceback (most recent call last):
   File "<stdin>", line 1, in ?
ValueError: multi
   line
detail
```

The last three lines (starting with `ValueError`) are compared against the exception’s type and detail, and the rest are ignored.

*Changed in version 2.4:* Previous versions were unable to handle multi-line exception details.

Best practice is to omit the traceback stack, unless it adds significant documentation value to the example. So the last example is probably better as:
```python
>>> raise ValueError('multi
line\ndetail')
Traceback (most recent call last):
...
ValueError: multi
  line
detail
```

Note that tracebacks are treated very specially. In particular, in the rewritten example, the use of `...` is independent of doctest’s **ELLIPSIS** option. The ellipsis in that example could be left out, or could just as well be three (or three hundred) commas or digits, or an indented transcript of a Monty Python skit.

Some details you should read once, but won’t need to remember:

- Doctest can’t guess whether your expected output came from an exception traceback or from ordinary printing. So, e.g., an example that expects `ValueError: 42 is prime` will pass whether `ValueError` is actually raised or if the example merely prints that traceback text. In practice, ordinary output rarely begins with a traceback header line, so this doesn’t create real problems.

- Each line of the traceback stack (if present) must be indented further than the first line of the example, or start with a non-alphanumeric character. The first line following the traceback header indented the same and starting with an alphanumeric is taken to be the start of the exception detail. Of course this does the right thing for genuine tracebacks.

- When the **IGNORE_EXCEPTION_DETAIL** doctest option is specified, everything following the leftmost colon and any module information in the exception name is ignored.

- The interactive shell omits the traceback header line for some **SyntaxError**s. But doctest uses the traceback header line to
distinguish exceptions from non-exceptions. So in the rare case where you need to test a `SyntaxError` that omits the traceback header, you will need to manually add the traceback header line to your test example.

- For some `SyntaxError`s, Python displays the character position of the syntax error, using a `^` marker:

```python
>>> 1 1
   File "<stdin>", line 1
   1 1
   ^
SyntaxError: invalid syntax
```

Since the lines showing the position of the error come before the exception type and detail, they are not checked by doctest. For example, the following test would pass, even though it puts the `^` marker in the wrong location:

```python
>>> 1 1
   File "<stdin>", line 1
   1 1
   ^
SyntaxError: invalid syntax
```

### 25.2.3.5. Option Flags

A number of option flags control various aspects of doctest’s behavior. Symbolic names for the flags are supplied as module constants, which can be or’ed together and passed to various functions. The names can also be used in doctest directives.

The first group of options define test semantics, controlling aspects of how doctest decides whether actual output matches an example’s expected output:

- `doctest.DONT_ACCEPT_TRUE_FOR_1`
By default, if an expected output block contains just 1, an actual output block containing just 1 or just True is considered to be a match, and similarly for 0 versus False. When DONT_ACCEPT_TRUE_FOR_1 is specified, neither substitution is allowed. The default behavior caters to that Python changed the return type of many functions from integer to boolean; doctests expecting “little integer” output still work in these cases. This option will probably go away, but not for several years.

**doctest. DONT_ACCEPT_BLANKLINE**

By default, if an expected output block contains a line containing only the string <BLANKLINE>, then that line will match a blank line in the actual output. Because a genuinely blank line delimits the expected output, this is the only way to communicate that a blank line is expected. When DONT_ACCEPT_BLANKLINE is specified, this substitution is not allowed.

**doctest. NORMALIZE_WHITESPACE**

When specified, all sequences of whitespace (blanks and newlines) are treated as equal. Any sequence of whitespace within the expected output will match any sequence of whitespace within the actual output. By default, whitespace must match exactly. NORMALIZE_WHITESPACE is especially useful when a line of expected output is very long, and you want to wrap it across multiple lines in your source.

**doctest. ELLIPSIS**

When specified, an ellipsis marker (…) in the expected output can match any substring in the actual output. This includes substrings that span line boundaries, and empty substrings, so it’s best to keep usage of this simple. Complicated uses can lead to the same kinds of “oops, it matched too much!” surprises that .* is prone to in regular expressions.
doctest. **IGNORE_EXCEPTION_DETAIL**

When specified, an example that expects an exception passes if an exception of the expected type is raised, even if the exception detail does not match. For example, an example expecting `ValueError: 42` will pass if the actual exception raised is `ValueError: 3*14`, but will fail, e.g., if `TypeError` is raised.

It will also ignore the module name used in Python 3 doctest reports. Hence both of these variations will work with the flag specified, regardless of whether the test is run under Python 2.7 or Python 3.2 (or later versions):

```python
>>> raise CustomError('message')
Traceback (most recent call last):
  CustomError: message
```

```python
>>> raise CustomError('message')
Traceback (most recent call last):
  my_module.CustomError: message
```

Note that **ELLIPSIS** can also be used to ignore the details of the exception message, but such a test may still fail based on whether or not the module details are printed as part of the exception name. Using **IGNORE_EXCEPTION_DETAIL** and the details from Python 2.3 is also the only clear way to write a doctest that doesn't care about the exception detail yet continues to pass under Python 2.3 or earlier (those releases do not support doctest directives and ignore them as irrelevant comments). For example:

```python
>>> (1, 2)[3] = 'moo'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

passes under Python 2.3 and later Python versions with the flag specified, even though the detail changed in Python 2.4 to say
“does not” instead of “doesn’t”.

Changed in version 2.7: `IGNORE_EXCEPTION_DETAIL` now also ignores any information relating to the module containing the exception under test.

doctest.SKIP
When specified, do not run the example at all. This can be useful in contexts where doctest examples serve as both documentation and test cases, and an example should be included for documentation purposes, but should not be checked. E.g., the example’s output might be random; or the example might depend on resources which would be unavailable to the test driver.

The SKIP flag can also be used for temporarily “commenting out” examples.

New in version 2.5.

doctest.COMPARISON_FLAGS
A bitmask or’ing together all the comparison flags above.

The second group of options controls how test failures are reported:

doctest.REPORT_UDIFF
When specified, failures that involve multi-line expected and actual outputs are displayed using a unified diff.

doctest.REPORT_CDIFF
When specified, failures that involve multi-line expected and actual outputs will be displayed using a context diff.

doctest.REPORT_NDIFF
When specified, differences are computed by `difflib.Differ`, using the same algorithm as the popular `ndiff.py` utility. This is the only method that marks differences within lines as well as
across lines. For example, if a line of expected output contains digit 1 where actual output contains letter l, a line is inserted with a caret marking the mismatching column positions.

**doctest.** REPORT_ONLY_FIRST_FAILURE

When specified, display the first failing example in each doctest, but suppress output for all remaining examples. This will prevent doctest from reporting correct examples that break because of earlier failures; but it might also hide incorrect examples that fail independently of the first failure. When **REPORT_ONLY_FIRST_FAILURE** is specified, the remaining examples are still run, and still count towards the total number of failures reported; only the output is suppressed.

**doctest.** REPORTING_FLAGS

A bitmask or’ing together all the reporting flags above.

*New in version 2.4:* The constants **DONT_ACCEPT_BLANKLINE**, **NORMALIZE_WHITESPACE**, **ELLIPSIS**, **IGNORE_EXCEPTION_DETAIL**, **REPORT_UDIFF**, **REPORT_CDIFF**, **REPORT_NDIFF**, **REPORT_ONLY_FIRST_FAILURE**, **COMPARISON_FLAGS** and **REPORTING_FLAGS** were added.

There’s also a way to register new option flag names, although this isn’t useful unless you intend to extend doctest internals via subclassing:

**doctest.** register_optionflag(name)

Create a new option flag with a given name, and return the new flag’s integer value. **register_optionflag()** can be used when subclassing **OutputChecker** or **DocTestRunner** to create new options that are supported by your subclasses. **register_optionflag()** should always be called using the following idiom:
MY_FLAG = register_optionflag('MY_FLAG')

New in version 2.4.

25.2.3.6. Directives

Doctest directives may be used to modify the option flags for an individual example. Doctest directives are special Python comments following an example's source code:

```
directive  ::=  "#"  "doctest:"  directive_options
directive_options  ::=  directive_option  ("",""  directive_option (",""  directive_option)
directive_option  ::=  on_or_off  directive_option_name
on_or_off  ::=  "+"  |  "-"
directive_option_name  ::=  "DONT_ACCEPT_BLANKLINE"  |  "NORMALIZE_WHITESPACE"  |  ...
```

Whitespace is not allowed between the + or - and the directive option name. The directive option name can be any of the option flag names explained above.

An example's doctest directives modify doctest's behavior for that single example. Use + to enable the named behavior, or - to disable it.

For example, this test passes:

```python
>>> print range(20) # doctest: +NORMALIZE_WHITESPACE
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Without the directive it would fail, both because the actual output doesn't have two blanks before the single-digit list elements, and because the actual output is on a single line. This test also passes, and also requires a directive to do so:

```python
>>> print range(20) # doctest: +ELLIPSIS
[0, 1, ..., 18, 19]
```
Multiple directives can be used on a single physical line, separated by commas:

```python
>>> print range(20)  # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
[0, 1, ..., 18, 19]
```

If multiple directive comments are used for a single example, then they are combined:

```python
>>> print range(20)  # doctest: +ELLIPSIS
...  # doctest: +NORMALIZE_WHITESPACE
[0, 1, ..., 18, 19]
```

As the previous example shows, you can add `...` lines to your example containing only directives. This can be useful when an example is too long for a directive to comfortably fit on the same line:

```python
>>> print range(5) + range(10, 20) + range(30, 40) + range(50, 60)
...  # doctest: +ELLIPSIS
[0, ..., 4, 10, ..., 19, 30, ..., 39, 50, ..., 59]
```

Note that since all options are disabled by default, and directives apply only to the example they appear in, enabling options (via `+` in a directive) is usually the only meaningful choice. However, option flags can also be passed to functions that run doctests, establishing different defaults. In such cases, disabling an option via `-` in a directive can be useful.

*New in version 2.4:* Support for doctest directives was added.

### 25.2.3.7. Warnings

`doctest` is serious about requiring exact matches in expected output. If even a single character doesn't match, the test fails. This will
probably surprise you a few times, as you learn exactly what Python
does and doesn't guarantee about output. For example, when
printing a dict, Python doesn't guarantee that the key-value pairs will
be printed in any particular order, so a test like

```python
>>> foo()
{'Hermione': 'hippogryph', 'Harry': 'broomstick'}
```

is vulnerable! One workaround is to do

```python
>>> foo() == {'Hermione': 'hippogryph', 'Harry': 'broomstick'}
True
```

instead. Another is to do

```python
>>> d = foo().items()
>>> d.sort()
>>> d
[('Harry', 'broomstick'), ('Hermione', 'hippogryph')]
```

There are others, but you get the idea.

Another bad idea is to print things that embed an object address, like

```python
>>> id(1.0)  # certain to fail some of the time
7948648
>>> class C: pass
>>> C()      # the default repr() for instances embeds an address
<__main__.C instance at 0x00AC18F0>
```

The **ELLIPSIS** directive gives a nice approach for the last example:

```python
>>> C()  # doctest: +ELLIPSIS
<__main__.C instance at 0x...>
```

Floating-point numbers are also subject to small output variations
across platforms, because Python defers to the platform C library for
float formatting, and C libraries vary widely in quality here.
Numbers of the form \( \frac{1}{2.**J} \) are safe across all platforms, and I often contrive doctest examples to produce numbers of that form:

Simple fractions are also easier for people to understand, and that makes for better documentation.
25.2.4. Basic API

The functions `testmod()` and `testfile()` provide a simple interface to doctest that should be sufficient for most basic uses. For a less formal introduction to these two functions, see sections Simple Usage: Checking Examples in Docstrings and Simple Usage: Checking Examples in a Text File.

```
doctest.testfile(filename[, module_relative][, name][, package][, globs][, verbose][, report][, optionflags][, extraglobs][, raise_on_error] [, parser][, encoding])
```

All arguments except `filename` are optional, and should be specified in keyword form.

Test examples in the file named `filename`. Return `(failure_count, test_count)`.

Optional argument `module_relative` specifies how the filename should be interpreted:

- If `module_relative` is `True` (the default), then `filename` specifies an OS-independent module-relative path. By default, this path is relative to the calling module’s directory; but if the `package` argument is specified, then it is relative to that package. To ensure OS-independence, `filename` should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If `module_relative` is `False`, then `filename` specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument `name` gives the name of the test; by default, or if `None`, `os.path.basename(filename)` is used.
Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for a module-relative filename. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is *False*.

Optional argument *globs* gives a dict to be used as the globals when executing examples. A new shallow copy of this dict is created for the doctest, so its examples start with a clean slate. By default, or if *None*, a new empty dict is used.

Optional argument *extraglobs* gives a dict merged into the globals used to execute examples. This works like *dict.update()*: if *globs* and *extraglobs* have a common key, the associated value in *extraglobs* appears in the combined dict. By default, or if *None*, no extra globals are used. This is an advanced feature that allows parameterization of doctests. For example, a doctest can be written for a base class, using a generic name for the class, then reused to test any number of subclasses by passing an *extraglobs* dict mapping the generic name to the subclass to be tested.

Optional argument *verbose* prints lots of stuff if true, and prints only failures if false; by default, or if *None*, it’s true if and only if *'-v'* is in *sys.argv*.

Optional argument *report* prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else the summary is very brief (in fact, empty if all tests passed).

Optional argument *optionflags* or's together option flags. See section *Option Flags*. 
Optional argument *raise_on_error* defaults to false. If true, an exception is raised upon the first failure or unexpected exception in an example. This allows failures to be post-mortem debugged. Default behavior is to continue running examples.

Optional argument *parser* specifies a *DocTestParser* (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., *DocTestParser()*).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

*New in version 2.4.*

*Changed in version 2.5:* The parameter *encoding* was added.

```python
doctest.testmod([m][, name][, globs][, verbose][, report][, optionflags][, extraglobs][, raise_on_error][, exclude_empty])
```

All arguments are optional, and all except for *m* should be specified in keyword form.

Test examples in docstrings in functions and classes reachable from module *m* (or module *__main__* if *m* is not supplied or is *None*), starting with *m.__doc__*.

Also test examples reachable from dict *m.__test__*, if it exists and is not *None*. *m.__test__* maps names (strings) to functions, classes and strings; function and class docstrings are searched for examples; strings are searched directly, as if they were docstrings.

Only docstrings attached to objects belonging to module *m* are searched.

Return (*failure_count*, *test_count*).
Optional argument *name* gives the name of the module; by default, or if `None, m.__name__` is used.

Optional argument *exclude_empty* defaults to false. If true, objects for which no doctests are found are excluded from consideration. The default is a backward compatibility hack, so that code still using `doctest.master.summarize()` in conjunction with `testmod()` continues to get output for objects with no tests. The *exclude_empty* argument to the newer `DocTestFinder` constructor defaults to true.

Optional arguments *extraglobs, verbose, report, optionflags, raise_on_error, and globs* are the same as for function `testfile()` above, except that *globs* defaults to `m.__dict__`.

*Changed in version 2.3:* The parameter *optionflags* was added.

*Changed in version 2.4:* The parameters *extraglobs, raise_on_error* and *exclude_empty* were added.

*Changed in version 2.5:* The optional argument *isprivate*, deprecated in 2.4, was removed.

```python
doctest.run_docstring_examples(f, globs[, verbose][, name][, compileflags][, optionflags])
```

Test examples associated with object *f*, for example, *f* may be a string, a module, a function, or a class object.

A shallow copy of dictionary argument *globs* is used for the execution context.

Optional argument *name* is used in failure messages, and defaults to "NoName".

If optional argument *verbose* is true, output is generated even if there are no failures. By default, output is generated only in case
of an example failure.

Optional argument `compileflags` gives the set of flags that should be used by the Python compiler when running the examples. By default, or if `None`, flags are deduced corresponding to the set of future features found in `globs`.

Optional argument `optionflags` works as for function `testfile()` above.
25.2.5. Unittest API

As your collection of doctest'ed modules grows, you'll want a way to run all their doctests systematically. Prior to Python 2.4, `doctest` had a barely documented `Tester` class that supplied a rudimentary way to combine doctests from multiple modules. `Tester` was feeble, and in practice most serious Python testing frameworks build on the `unittest` module, which supplies many flexible ways to combine tests from multiple sources. So, in Python 2.4, `doctest`'s `Tester` class is deprecated, and `doctest` provides two functions that can be used to create `unittest` test suites from modules and text files containing doctests. To integrate with `unittest` test discovery, include a `load_tests()` function in your test module:

```python
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

There are two main functions for creating `unittest.TestSuite` instances from text files and modules with doctests:

`doctest.DocFileSuite(*paths, [module_relative][, package][, setUp] [, tearDown][, globs][, optionflags][, parser][, encoding])`

Convert doctest tests from one or more text files to a `unittest.TestSuite`.

The returned `unittest.TestSuite` is to be run by the unittest framework and runs the interactive examples in each file. If an example in any file fails, then the synthesized unit test fails, and a
failureException exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Pass one or more paths (as strings) to text files to be examined.

Options may be provided as keyword arguments:

Optional argument module_relative specifies how the filenames in paths should be interpreted:

- If module_relative is True (the default), then each filename in paths specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the package argument is specified, then it is relative to that package. To ensure OS-independence, each filename should use / characters to separate path segments, and may not be an absolute path (i.e., it may not begin with /).
- If module_relative is False, then each filename in paths specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument package is a Python package or the name of a Python package whose directory should be used as the base directory for module-relative filenames in paths. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify package if module_relative is False.

Optional argument setUp specifies a set-up function for the test suite. This is called before running the tests in each file. The setUp function will be passed a DocTest object. The setUp function can access the test globals as the globs attribute of the test passed.
Optional argument `tearDown` specifies a tear-down function for the test suite. This is called after running the tests in each file. The `tearDown` function will be passed a `DocTest` object. The `setUp` function can access the test globals as the `globs` attribute of the test passed.

Optional argument `globs` is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, `globs` is a new empty dictionary.

Optional argument `optionflags` specifies the default doctest options for the tests, created by or-ing together individual option flags. See section Option Flags. See function `set_unittest_reportflags()` below for a better way to set reporting options.

Optional argument `parser` specifies a `DocTestParser` (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument `encoding` specifies an encoding that should be used to convert the file to unicode.

*New in version 2.4.*

*Changed in version 2.5:* The global `__file__` was added to the globals provided to doctests loaded from a text file using `DocFileSuite()`.

*Changed in version 2.5:* The parameter `encoding` was added.

**Note:** Unlike `testmod()` and `DocTestFinder`, this function raises a `ValueError` if `module` contains no docstrings. You can prevent this error by passing a `DocTestFinder` instance as the `test_finder` argument with its `exclude_empty` keyword argument
**set to False:**

```python
>>> finder = doctest.DocTestFinder(exclude_empty=False)
>>> suite = doctest.DocTestSuite(test_finder=finder)
```

doctest.**DocTestSuite**([module][, globs][, extraglobs][, test_finder][, setUp][, tearDown][, checker])

Convert doctest tests for a module to a **unittest.TestSuite**.

The returned **unittest.TestSuite** is to be run by the unittest framework and runs each doctest in the module. If any of the doctests fail, then the synthesized unit test fails, and a **failureException** exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Optional argument **module** provides the module to be tested. It can be a module object or a (possibly dotted) module name. If not specified, the module calling this function is used.

Optional argument **globs** is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, **globs** is a new empty dictionary.

Optional argument **extraglobs** specifies an extra set of global variables, which is merged into **globs**. By default, no extra globals are used.

Optional argument **test_finder** is the **DocTestFinder** object (or a drop-in replacement) that is used to extract doctests from the module.

Optional arguments **setUp**, **tearDown**, and **optionflags** are the same as for function **DocFileSuite()** above.
New in version 2.3.

Changed in version 2.4: The parameters `globs`, `extraglobs`, `test_finder`, `setUp`, `tearDown`, and `optionflags` were added; this function now uses the same search technique as `testmod()`.

Under the covers, `DocTestSuite()` creates a `unittest.TestSuite` out of `doctest.DocTestCase` instances, and `DocTestCase` is a subclass of `unittest.TestCase`. `DocTestCase` isn’t documented here (it’s an internal detail), but studying its code can answer questions about the exact details of `unittest` integration.

Similarly, `DocFileSuite()` creates a `unittest.TestSuite` out of `doctest.DocFileCase` instances, and `DocFileCase` is a subclass of `DocTestCase`.

So both ways of creating a `unittest.TestSuite` run instances of `DocTestCase`. This is important for a subtle reason: when you run `doctest` functions yourself, you can control the `doctest` options in use directly, by passing option flags to `doctest` functions. However, if you’re writing a `unittest` framework, `unittest` ultimately controls when and how tests get run. The framework author typically wants to control `doctest` reporting options (perhaps, e.g., specified by command line options), but there’s no way to pass options through `unittest` to `doctest` test runners.

For this reason, `doctest` also supports a notion of `doctest` reporting flags specific to `unittest` support, via this function:

```python
doctest.set_unittest_reportflags(flags)
```

Set the `doctest` reporting flags to use.

Argument `flags` or's together option flags. See section Option Flags. Only "reporting flags" can be used.
This is a module-global setting, and affects all future doctests run by module `unittest`: the `runTest()` method of `DocTestCase` looks at the option flags specified for the test case when the `DocTestCase` instance was constructed. If no reporting flags were specified (which is the typical and expected case), `doctest`'s `unittest` reporting flags are or’ed into the option flags, and the option flags so augmented are passed to the `DocTestRunner` instance created to run the doctest. If any reporting flags were specified when the `DocTestCase` instance was constructed, `doctest`'s `unittest` reporting flags are ignored.

The value of the `unittest` reporting flags in effect before the function was called is returned by the function.

*New in version 2.4.*
25.2.6. Advanced API

The basic API is a simple wrapper that’s intended to make doctest easy to use. It is fairly flexible, and should meet most users’ needs; however, if you require more fine-grained control over testing, or wish to extend doctest’s capabilities, then you should use the advanced API.

The advanced API revolves around two container classes, which are used to store the interactive examples extracted from doctest cases:

- **Example**: A single Python `statement`, paired with its expected output.
- **DocTest**: A collection of Examples, typically extracted from a single docstring or text file.

Additional processing classes are defined to find, parse, and run, and check doctest examples:

- **DocTestFinder**: Finds all docstrings in a given module, and uses a `DocTestParser` to create a `DocTest` from every docstring that contains interactive examples.
- **DocTestParser**: Creates a `DocTest` object from a string (such as an object’s docstring).
- **DocTestRunner**: Executes the examples in a `DocTest`, and uses an `OutputChecker` to verify their output.
- **OutputChecker**: Compares the actual output from a doctest example with the expected output, and decides whether they match.

The relationships among these processing classes are summarized in the following diagram:
25.2.6.1. DocTest Objects

```python
class doctest.DocTest(examples, globs, name, filename, lineno, docstring)
```

A collection of doctest examples that should be run in a single namespace. The constructor arguments are used to initialize the attributes of the same names.

*New in version 2.4.*

**DocTest** defines the following attributes. They are initialized by the constructor, and should not be modified directly.

**examples**

A list of **Example** objects encoding the individual interactive Python examples that should be run by this test.

**globs**

The namespace (aka globals) that the examples should be run in. This is a dictionary mapping names to values. Any changes to the namespace made by the examples (such as binding new variables) will be reflected in **globs** after the test is run.

**name**

A string name identifying the **DocTest**. Typically, this is the name of the object or file that the test was extracted from.
**filename**

The name of the file that this DocTest was extracted from; or `None` if the filename is unknown, or if the DocTest was not extracted from a file.

**lineno**

The line number within `filename` where this DocTest begins, or `None` if the line number is unavailable. This line number is zero-based with respect to the beginning of the file.

**docstring**

The string that the test was extracted from, or `None` if the string is unavailable, or if the test was not extracted from a string.

---

### 25.2.6.2. Example Objects

```python
class doctest.Example(source, want[, exc_msg][, lineno][, indent][, options])
```

A single interactive example, consisting of a Python statement and its expected output. The constructor arguments are used to initialize the attributes of the same names.

*New in version 2.4.*

**Example** defines the following attributes. They are initialized by the constructor, and should not be modified directly.

**source**

A string containing the example's source code. This source code consists of a single Python statement, and always ends with a newline; the constructor adds a newline when necessary.

**want**
The expected output from running the example’s source code (either from stdout, or a traceback in case of exception). `want` ends with a newline unless no output is expected, in which case it’s an empty string. The constructor adds a newline when necessary.

**exc_msg**
The exception message generated by the example, if the example is expected to generate an exception; or `None` if it is not expected to generate an exception. This exception message is compared against the return value of `traceback.format_exception_only()`. `exc_msg` ends with a newline unless it's `None`. The constructor adds a newline if needed.

**lineno**
The line number within the string containing this example where the example begins. This line number is zero-based with respect to the beginning of the containing string.

**indent**
The example's indentation in the containing string, i.e., the number of space characters that precede the example's first prompt.

**options**
A dictionary mapping from option flags to `True` or `False`, which is used to override default options for this example. Any option flags not contained in this dictionary are left at their default value (as specified by the `DocTestRunner`'s `optionflags`). By default, no options are set.

### 25.2.6.3. DocTestFinder objects

```python
class doctest.DocTestFinder([verbose][, parser][, recurse][,]
```
A processing class used to extract the DocTest s that are relevant to a given object, from its docstring and the docstrings of its contained objects. DocTest s can currently be extracted from the following object types: modules, functions, classes, methods, staticmethods, classmethods, and properties.

The optional argument verbose can be used to display the objects searched by the finder. It defaults to False (no output).

The optional argument parser specifies the DocTestParser object (or a drop-in replacement) that is used to extract doctests from docstrings.

If the optional argument recurse is false, then DocTestFinder.find() will only examine the given object, and not any contained objects.

If the optional argument exclude_empty is false, then DocTestFinder.find() will include tests for objects with empty docstrings.

New in version 2.4.

DocTestFinder defines the following method:

\textbf{find}(obj[, name][, module][, globs][, extraglobs])

Return a list of the DocTest s that are defined by obj’s docstring, or by any of its contained objects’ docstrings.

The optional argument name specifies the object’s name; this name will be used to construct names for the returned DocTest s. If name is not specified, then obj.__name__ is used.

The optional parameter module is the module that contains
the given object. If the module is not specified or is None, then the test finder will attempt to automatically determine the correct module. The object’s module is used:

- As a default namespace, if globs is not specified.
- To prevent the DocTestFinder from extracting DocTests from objects that are imported from other modules. (Contained objects with modules other than module are ignored.)
- To find the name of the file containing the object.
- To help find the line number of the object within its file.

If module is False, no attempt to find the module will be made. This is obscure, of use mostly in testing doctest itself: if module is False, or is None but cannot be found automatically, then all objects are considered to belong to the (non-existent) module, so all contained objects will (recursively) be searched for doctests.

The globals for each DocTest is formed by combining globs and extraglobs (bindings in extraglobs override bindings in globs). A new shallow copy of the globals dictionary is created for each DocTest. If globs is not specified, then it defaults to the module’s __dict__, if specified, or {} otherwise. If extraglobs is not specified, then it defaults to {}.

### 25.2.6.4. DocTestParser objects

**class doctest.DocTestParser**

A processing class used to extract interactive examples from a string, and use them to create a DocTest object.

New in version 2.4.

DocTestParser defines the following methods:
**get_doctest** *(string, globs, name, filename, lineno)*

Extract all doctest examples from the given string, and collect them into a **DocTest** object.

* globs, name, filename, and lineno are attributes for the new **DocTest** object. See the documentation for **DocTest** for more information.

**get_examples** *(string[, name])*  
Extract all doctest examples from the given string, and return them as a list of **Example** objects. Line numbers are 0-based. The optional argument name is a name identifying this string, and is only used for error messages.

**parse** *(string[, name])*  
Divide the given string into examples and intervening text, and return them as a list of alternating **Examples** and strings. Line numbers for the **Examples** are 0-based. The optional argument name is a name identifying this string, and is only used for error messages.

### 25.2.6.5. DocTestRunner objects

**class** doctest. **DocTestRunner** ([checker[, verbose[, optionflags]]])

A processing class used to execute and verify the interactive examples in a **DocTest**.

The comparison between expected outputs and actual outputs is done by an **OutputChecker**. This comparison may be customized with a number of option flags; see section **Option Flags** for more information. If the option flags are insufficient, then the comparison may also be customized by passing a subclass of
OutputChecker to the constructor.

The test runner’s display output can be controlled in two ways. First, an output function can be passed to TestRunner.run(); this function will be called with strings that should be displayed. It defaults to sys.stdout.write. If capturing the output is not sufficient, then the display output can be also customized by subclassing DocTestRunner, and overriding the methods report_start(), report_success(), report_unexpected_exception(), and report_failure().

The optional keyword argument checker specifies the OutputChecker object (or drop-in replacement) that should be used to compare the expected outputs to the actual outputs of doctest examples.

The optional keyword argument verbose controls the DocTestRunner’s verbosity. If verbose is True, then information is printed about each example, as it is run. If verbose is False, then only failures are printed. If verbose is unspecified, or None, then verbose output is used iff the command-line switch -v is used.

The optional keyword argument optionflags can be used to control how the test runner compares expected output to actual output, and how it displays failures. For more information, see section Option Flags.

New in version 2.4.

DocTestParser defines the following methods:

report_start(out, test, example)  
Report that the test runner is about to process the given example. This method is provided to allow subclasses of
**DocTestRunner** to customize their output; it should not be called directly.

*example* is the example about to be processed. *test* is the test containing *example*. *out* is the output function that was passed to **DocTestRunner.run()**.

**report_success**(*out, test, example, got*)

Report that the given example ran successfully. This method is provided to allow subclasses of **DocTestRunner** to customize their output; it should not be called directly.

*example* is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to **DocTestRunner.run()**.

**report_failure**(*out, test, example, got*)

Report that the given example failed. This method is provided to allow subclasses of **DocTestRunner** to customize their output; it should not be called directly.

*example* is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to **DocTestRunner.run()**.

**report_unexpected_exception**(*out, test, example, exc_info*)

Report that the given example raised an unexpected exception. This method is provided to allow subclasses of **DocTestRunner** to customize their output; it should not be called directly.

*example* is the example about to be processed. *exc_info* is a tuple containing information about the unexpected exception
(as returned by `sys.exc_info()`). `test` is the test containing example. `out` is the output function that was passed to `DocTestRunner.run()`.

`run(test[, compileflags][, out][, clear_globs])`

Run the examples in `test` (a `DocTest` object), and display the results using the writer function `out`.

The examples are run in the namespace `test.globs`. If `clear_globs` is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use `clear_globs=False`.

`compileflags` gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to `globs`.

The output of each example is checked using the `DocTestRunner`'s output checker, and the results are formatted by the `DocTestRunner.report_*()` methods.

`summarize([verbose])`

Print a summary of all the test cases that have been run by this `DocTestRunner`, and return a named tuple `TestResults(failed, attempted)`.

The optional `verbose` argument controls how detailed the summary is. If the verbosity is not specified, then the `DocTestRunner`'s verbosity is used.

*Changed in version 2.6*: Use a named tuple.
25.2.6.6. OutputChecker objects

class doctest.OutputChecker

A class used to check whether the actual output from a doctest example matches the expected output. OutputChecker defines two methods: check_output(), which compares a given pair of outputs, and returns true if they match; and output_difference(), which returns a string describing the differences between two outputs.

New in version 2.4.

OutputChecker defines the following methods:

check_output(want, got, optionflags)

Return True iff the actual output from an example (got) matches the expected output (want). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See section Option Flags for more information about option flags.

output_difference(example, got, optionflags)

Return a string describing the differences between the expected output for a given example (example) and the actual output (got). optionflags is the set of option flags used to compare want and got.
25.2.7. Debugging

Doctest provides several mechanisms for debugging doctest examples:

- Several functions convert doctests to executable Python programs, which can be run under the Python debugger, `pdb`.

- The `DebugRunner` class is a subclass of `DocTestRunner` that raises an exception for the first failing example, containing information about that example. This information can be used to perform post-mortem debugging on the example.

- The `unittest` cases generated by `DocTestSuite()` support the `debug()` method defined by `unittest.TestCase`.

- You can add a call to `pdb.set_trace()` in a doctest example, and you'll drop into the Python debugger when that line is executed. Then you can inspect current values of variables, and so on. For example, suppose `a.py` contains just this module docstring:

```python
""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print x+3
...     import pdb; pdb.set_trace()
>>> f(3)
g
""
```

Then an interactive Python session may look like this:

```python
>>> import a, doctest
>>> doctest.testmod(a)--Return--
> <doctest a[1]>(3)g()--None
```
import doctest

Changed in version 2.4: The ability to use `pdb.set_trace()` usefully inside doctests was added.

Functions that convert doctests to Python code, and possibly run the synthesized code under the debugger:

```
doctest.script_from_examples(s)
```

Convert text with examples to a script.

Argument `s` is a string containing doctest examples. The string is converted to a Python script, where doctest examples in `s` are converted to regular code, and everything else is converted to Python comments. The generated script is returned as a string. For example,

```python
import doctest
```
print doctest.script_from_examples(r""
    Set \textit{x} and \textit{y} to 1 and 2.
    >>> x, y = 1, 2

    \texttt{Print their sum:}
    >>> print x+y
    3
"")

displays:

```
# Set \textit{x} and \textit{y} to 1 and 2.
x, y = 1, 2
#
# \texttt{Print their sum:}
print x+y
# \texttt{Expected:}
## 3
```

This function is used internally by other functions (see below), but can also be useful when you want to transform an interactive Python session into a Python script.

\textit{New in version 2.4.}

doctest.\texttt{testsource}\texttt{(module, name)}

Convert the doctest for an object to a script.

Argument \textit{module} is a module object, or dotted name of a module, containing the object whose doctests are of interest. Argument \textit{name} is the name (within the module) of the object with the doctests of interest. The result is a string, containing the object's docstring converted to a Python script, as described for \texttt{script_from_examples()} above. For example, if module \texttt{a.py} contains a top-level function \texttt{f()}, then

```python
import a, doctest
print doctest.testsource(a, "a.f")
```
prints a script version of function \( f() \)'s docstring, with doctests converted to code, and the rest placed in comments.

*New in version 2.3.*

```
>>> func = lambda x: x
>>> doctest.debug(f, 'print(f(2))')
```

**doctest.debug** *(module, name[, pm]*)

Debug the doctests for an object.

The *module* and *name* arguments are the same as for function `testsource()` above. The synthesized Python script for the named object’s docstring is written to a temporary file, and then that file is run under the control of the Python debugger, `pdb`.

A shallow copy of `module.__dict__` is used for both local and global execution context.

Optional argument *pm* controls whether post-mortem debugging is used. If *pm* has a true value, the script file is run directly, and the debugger gets involved only if the script terminates via raising an unhandled exception. If it does, then post-mortem debugging is invoked, via `pdb.post_mortem()`, passing the traceback object from the unhandled exception. If *pm* is not specified, or is false, the script is run under the debugger from the start, via passing an appropriate `execfile()` call to `pdb.run()`.

*New in version 2.3.*

*Changed in version 2.4:* The *pm* argument was added.

```
>>> doctest.debug_src('print(x)', pm)
```

**doctest.debug_src** *(src[, pm][, globs]*)

Debug the doctests in a string.

This is like function `debug()` above, except that a string containing doctest examples is specified directly, via the *src* argument.
Optional argument \textit{pm} has the same meaning as in function \texttt{debug()} above.

Optional argument \textit{ globs} gives a dictionary to use as both local and global execution context. If not specified, or \texttt{None}, an empty dictionary is used. If specified, a shallow copy of the dictionary is used.

\textit{New in version 2.4.}

The \texttt{DebugRunner} class, and the special exceptions it may raise, are of most interest to testing framework authors, and will only be sketched here. See the source code, and especially \texttt{DebugRunner}'s docstring (which is a doctest!) for more details:

\begin{verbatim}
class doctest. DebugRunner([checker], [verbose], [optionflags])
  A subclass of DocTestRunner that raises an exception as soon as a failure is encountered. If an unexpected exception occurs, an UnexpectedException exception is raised, containing the test, the example, and the original exception. If the output doesn't match, then a DocTestFailure exception is raised, containing the test, the example, and the actual output.

  For information about the constructor parameters and methods, see the documentation for DocTestRunner in section Advanced API.

There are two exceptions that may be raised by DebugRunner instances:

exception doctest. DocTestFailure(test, example, got)
  An exception raised by DocTestRunner to signal that a doctest example's actual output did not match its expected output. The constructor arguments are used to initialize the attributes of the same names.
\end{verbatim}
**DocTestFailure** defines the following attributes:

DocTestFailure.*test*

The **DocTest** object that was being run when the example failed.

DocTestFailure.*example*

The **Example** that failed.

DocTestFailure.*got*

The example's actual output.

*exception* doctest.**UnexpectedException**(test, example, exc_info)

An exception raised by **DocTestRunner** to signal that a doctest example raised an unexpected exception. The constructor arguments are used to initialize the attributes of the same names.

**UnexpectedException** defines the following attributes:

UnexpectedException.*test*

The **DocTest** object that was being run when the example failed.

UnexpectedException.*example*

The **Example** that failed.

UnexpectedException.*exc_info*

A tuple containing information about the unexpected exception, as returned by `sys.exc_info()`.
As mentioned in the introduction, doctest has grown to have three primary uses:

1. Checking examples in docstrings.
2. Regression testing.
3. Executable documentation / literate testing.

These uses have different requirements, and it is important to distinguish them. In particular, filling your docstrings with obscure test cases makes for bad documentation.

When writing a docstring, choose docstring examples with care. There's an art to this that needs to be learned—it may not be natural at first. Examples should add genuine value to the documentation. A good example can often be worth many words. If done with care, the examples will be invaluable for your users, and will pay back the time it takes to collect them many times over as the years go by and things change. I'm still amazed at how often one of my doctest examples stops working after a “harmless” change.

Doctest also makes an excellent tool for regression testing, especially if you don’t skimp on explanatory text. By interleaving prose and examples, it becomes much easier to keep track of what’s actually being tested, and why. When a test fails, good prose can make it much easier to figure out what the problem is, and how it should be fixed. It’s true that you could write extensive comments in code-based testing, but few programmers do. Many have found that using doctest approaches instead leads to much clearer tests. Perhaps this is simply because doctest makes writing prose a little easier than writing code, while writing comments in code is a little harder. I think it goes deeper than just that: the natural attitude when writing a doctest-based test is that you want to explain the fine points
of your software, and illustrate them with examples. This in turn naturally leads to test files that start with the simplest features, and logically progress to complications and edge cases. A coherent narrative is the result, instead of a collection of isolated functions that test isolated bits of functionality seemingly at random. It’s a different attitude, and produces different results, blurring the distinction between testing and explaining.

Regression testing is best confined to dedicated objects or files. There are several options for organizing tests:

- Write text files containing test cases as interactive examples, and test the files using `testfile()` or `DocFileSuite()`. This is recommended, although is easiest to do for new projects, designed from the start to use doctest.
- Define functions named `_regrtest_topic` that consist of single docstrings, containing test cases for the named topics. These functions can be included in the same file as the module, or separated out into a separate test file.
- Define a `__test__` dictionary mapping from regression test topics to docstrings containing test cases.

When you have placed your tests in a module, the module can itself be the test runner. When a test fails, you can arrange for your test runner to re-run only the failing doctest while you debug the problem. Here is a minimal example of such a test runner:

```python
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.REPORT_ONLY_FIRST_FAILURE
    if len(sys.argv) > 1:
        name = sys.argv[1]
    if name in globals():
        obj = globals()[name]
    else:
        obj = __test__[name]
    doctest.run_docstring_examples(obj, globals(), name=name)
```
```python
else:
    fail, total = doctest.testmod(optionflags=flags)
    print("{} failures out of {} tests".format(fail, total))
```

**Footnotes**

[1] Examples containing both expected output and an exception are not supported. Trying to guess where one ends and the other begins is too error-prone, and that also makes for a confusing test.
25.3. unittest — Unit testing framework

New in version 2.1.

(If you are already familiar with the basic concepts of testing, you might want to skip to the list of assert methods.)

The Python unit testing framework, sometimes referred to as “PyUnit,” is a Python language version of JUnit, by Kent Beck and Erich Gamma. JUnit is, in turn, a Java version of Kent’s Smalltalk testing framework. Each is the de facto standard unit testing framework for its respective language.

unittest supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. The unittest module provides classes that make it easy to support these qualities for a set of tests.

To achieve this, unittest supports some important concepts:

test fixture

A test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case

A test case is the smallest unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, TestCase, which may be used to create new test cases.

test suite
A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

test runner

A test runner is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

The test case and test fixture concepts are supported through the TestCase and FunctionTestCase classes; the former should be used when creating new tests, and the latter can be used when integrating existing test code with a unittest-driven framework. When building test fixtures using TestCase, the setUp() and tearDown() methods can be overridden to provide initialization and cleanup for the fixture. With FunctionTestCase, existing functions can be passed to the constructor for these purposes. When the test is run, the fixture initialization is run first; if it succeeds, the cleanup method is run after the test has been executed, regardless of the outcome of the test. Each instance of the TestCase will only be used to run a single test method, so a new fixture is created for each test.

Test suites are implemented by the TestSuite class. This class allows individual tests and test suites to be aggregated; when the suite is executed, all tests added directly to the suite and in “child” test suites are run.

A test runner is an object that provides a single method, run(), which accepts a TestCase or TestSuite object as a parameter, and returns a result object. The class TestResult is provided for use as the result object. unittest provides the TextTestRunner as an example test runner which reports test results on the standard error stream by default. Alternate runners can be implemented for other environments (such as graphical environments) without any need to
derive from a specific class.

See also:

**Module doctest**

Another test-support module with a very different flavor.

**unittest2: A backport of new unittest features for Python 2.4-2.6**

Many new features were added to unittest in Python 2.7, including test discovery. unittest2 allows you to use these features with earlier versions of Python.

**Simple Smalltalk Testing: With Patterns**

Kent Beck's original paper on testing frameworks using the pattern shared by unittest.

**Nose and py.test**

Third-party unittest frameworks with a lighter-weight syntax for writing tests. For example, `assert func(10) == 42`.

**The Python Testing Tools Taxonomy**

An extensive list of Python testing tools including functional testing frameworks and mock object libraries.

**Testing in Python Mailing List**

A special-interest-group for discussion of testing, and testing tools, in Python.
25.3.1. Basic example

The `unittest` module provides a rich set of tools for constructing and running tests. This section demonstrates that a small subset of the tools suffice to meet the needs of most users.

Here is a short script to test three string methods:

```python
import unittest

class TestStringMethods(unittest.TestCase):
    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

A testcase is created by subclassing `unittest.TestCase`. The three individual tests are defined with methods whose names start with the letters `test`. This naming convention informs the test runner about which methods represent tests.

The crux of each test is a call to `assertEqual()` to check for an expected result; `assertTrue()` or `assertFalse()` to verify a condition; or `assertRaises()` to verify that a specific exception gets raised.
These methods are used instead of the `assert` statement so the test runner can accumulate all test results and produce a report.

The `setUp()` and `tearDown()` methods allow you to define instructions that will be executed before and after each test method. They are covered in more detail in the section Organizing test code.

The final block shows a simple way to run the tests. `unittest.main()` provides a command-line interface to the test script. When run from the command line, the above script produces an output that looks like this:

```python
...  
Ran 3 tests in 0.000s  
OK  
```

Instead of `unittest.main()`, there are other ways to run the tests with a finer level of control, less terse output, and no requirement to be run from the command line. For example, the last two lines may be replaced with:

```python
suite = unittest.TestLoader().loadTestsFromTestCase(TestStringMethods)  
suite = unittest.TextTestRunner(verbosity=2).run(suite)  
```

Running the revised script from the interpreter or another script produces the following output:

```python
test_isupper (__main__.TestStringMethods) ... ok  
test_split (__main__.TestStringMethods) ... ok  
test_upper (__main__.TestStringMethods) ... ok  

Ran 3 tests in 0.001s  
OK  
```
The above examples show the most commonly used `unittest` features which are sufficient to meet many everyday testing needs. The remainder of the documentation explores the full feature set from first principles.
25.3.2. Command-Line Interface

The unittest module can be used from the command line to run tests from modules, classes or even individual test methods:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

You can pass in a list with any combination of module names, and fully qualified class or method names.

You can run tests with more detail (higher verbosity) by passing in the `-v` flag:

```
python -m unittest -v test_module
```

For a list of all the command-line options:

```
python -m unittest -h
```

*Changed in version 2.7:* In earlier versions it was only possible to run individual test methods and not modules or classes.

25.3.2.1. Command-line options

`unittest` supports these command-line options:

**-b, --buffer**

The standard output and standard error streams are buffered during the test run. Output during a passing test is discarded. Output is echoed normally on test fail or error and is added to the failure messages.

**-c, --catch**
Control-C during the test run waits for the current test to end and then reports all the results so far. A second Control-C raises the normal KeyboardInterrupt exception.

See Signal Handling for the functions that provide this functionality.

-f, --failfast
Stop the test run on the first error or failure.

New in version 2.7: The command-line options -b, -c and -f were added.

The command line can also be used for test discovery, for running all of the tests in a project or just a subset.
25.3.3. Test Discovery

New in version 2.7.

Unittest supports simple test discovery. In order to be compatible with test discovery, all of the test files must be modules or packages importable from the top-level directory of the project (this means that their filenames must be valid identifiers).

Test discovery is implemented in `TestLoader.discover()`, but can also be used from the command line. The basic command-line usage is:

```
import os
os.chdir('project_directory')
python -m unittest discover
```

The `discover` sub-command has the following options:

- **-v, --verbose**
  Verbose output

- **-s, --start-directory directory**
  Directory to start discovery (. default)

- **-p, --pattern pattern**
  Pattern to match test files (test*.py default)

- **-t, --top-level-directory directory**
  Top level directory of project (defaults to start directory)

The `-s`, `-p`, and `-t` options can be passed in as positional arguments in that order. The following two command lines are equivalent:

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```
As well as being a path it is possible to pass a package name, for example `myproject.subpackage.test`, as the start directory. The package name you supply will then be imported and its location on the filesystem will be used as the start directory.

**Caution:** Test discovery loads tests by importing them. Once test discovery has found all the test files from the start directory you specify it turns the paths into package names to import. For example `foo/bar/baz.py` will be imported as `foo.bar.baz`.

If you have a package installed globally and attempt test discovery on a different copy of the package then the import *could* happen from the wrong place. If this happens test discovery will warn you and exit.

If you supply the start directory as a package name rather than a path to a directory then discover assumes that whichever location it imports from is the location you intended, so you will not get the warning.

Test modules and packages can customize test loading and discovery by through the `load_tests` protocol.
25.3.4. Organizing test code

The basic building blocks of unit testing are *test cases* — single scenarios that must be set up and checked for correctness. In *unittest*, test cases are represented by instances of *unittest*'s *TestCase* class. To make your own test cases you must write subclasses of *TestCase*, or use *FunctionTestCase*.

An instance of a *TestCase*-derived class is an object that can completely run a single test method, together with optional set-up and tidy-up code.

The testing code of a *TestCase* instance should be entirely self contained, such that it can be run either in isolation or in arbitrary combination with any number of other test cases.

The simplest *TestCase* subclass will simply override the *runTest()* method in order to perform specific testing code:

```python
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def runTest(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50), 'incorrect default size')
```

Note that in order to test something, we use one of the *assert*() methods provided by the *TestCase* base class. If the test fails, an exception will be raised, and *unittest* will identify the test case as a *failure*. Any other exceptions will be treated as *errors*. This helps you identify where the problem is: *failures* are caused by incorrect results - a 5 where you expected a 6. *Errors* are caused by incorrect code - e.g., a *TypeError* caused by an incorrect function call.
The way to run a test case will be described later. For now, note that to construct an instance of such a test case, we call its constructor without arguments:

```python
testCase = DefaultWidgetSizeTestCase()
```

Now, such test cases can be numerous, and their set-up can be repetitive. In the above case, constructing a `Widget` in each of 100 `Widget` test case subclasses would mean unsightly duplication.

Luckily, we can factor out such set-up code by implementing a method called `setUp()`, which the testing framework will automatically call for us when we run the test:

```python
import unittest
class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

class DefaultWidgetSizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        self.assertEqual(self.widget.size(), (50, 50), 'incorrect default size')

class WidgetResizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        self.widget.resize(100, 150)
        self.assertEqual(self.widget.size(), (100, 150), 'wrong size after resize')
```

If the `setUp()` method raises an exception while the test is running, the framework will consider the test to have suffered an error, and the `runTest()` method will not be executed.

Similarly, we can provide a `tearDown()` method that tidies up after the `runTest()` method has been run:

```python
import unittest
```
class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')
    def tearDown(self):
        self.widget.dispose()
        self.widget = None

If `setUp()` succeeded, the `tearDown()` method will be run whether `runTest()` succeeded or not.

Such a working environment for the testing code is called a fixture.

Often, many small test cases will use the same fixture. In this case, we would end up subclassing `SimpleWidgetTestCase` into many small one-method classes such as `DefaultWidgetSizeTestCase`. This is time-consuming and discouraging, so in the same vein as JUnit, `unittest` provides a simpler mechanism:

```python
import unittest
class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')
    def tearDown(self):
        self.widget.dispose()
        self.widget = None
    def test_default_size(self):
        self.assertEqual(self.widget.size(), (50, 50), 'incorrect default size')
    def test_resize(self):
        self.widget.resize(100, 150)
        self.assertEqual(self.widget.size(), (100, 150), 'wrong size after resize')
```

Here we have not provided a `runTest()` method, but have instead provided two different test methods. Class instances will now each
run one of the test_*() methods, with `self.widget` created and destroyed separately for each instance. When creating an instance we must specify the test method it is to run. We do this by passing the method name in the constructor:

```python
defaultSizeTestCase = WidgetTestCase('test_default_size')
resizeTestCase = WidgetTestCase('test_resize')
```

Test case instances are grouped together according to the features they test. `unittest` provides a mechanism for this: the test suite, represented by `unittest`'s `TestSuite` class:

```python
widgetTestSuite = unittest.TestSuite()
widgetTestSuite.addTest(WidgetTestCase('test_default_size'))
widgetTestSuite.addTest(WidgetTestCase('test_resize'))
```

For the ease of running tests, as we will see later, it is a good idea to provide in each test module a callable object that returns a pre-built test suite:

```python
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_size'))
    suite.addTest(WidgetTestCase('test_resize'))
    return suite
```

or even:

```python
def suite():
    tests = ['test_default_size', 'test_resize']

    return unittest.TestSuite(map(WidgetTestCase, tests))
```

Since it is a common pattern to create a `TestCase` subclass with many similarly named test functions, `unittest` provides a `TestLoader` class that can be used to automate the process of creating a test suite and populating it with individual tests. For example,
suite = unittest.TestLoader().loadTestsFromTestCase(WidgetTestCase)

will create a test suite that will run
WidgetTestCase.test_default_size() and
WidgetTestCase.test_resize. TestLoader uses the 'test' method
name prefix to identify test methods automatically.

Note that the order in which the various test cases will be run is
determined by sorting the test function names with respect to the
built-in ordering for strings.

Often it is desirable to group suites of test cases together, so as to
run tests for the whole system at once. This is easy, since TestSuite
instances can be added to a TestSuite just as TestCase instances
can be added to a TestSuite:

```python
suite1 = module1.TheTestSuite()
suite2 = module2.TheTestSuite()
alltests = unittest.TestSuite([suite1, suite2])
```

You can place the definitions of test cases and test suites in the
same modules as the code they are to test (such as widget.py), but
there are several advantages to placing the test code in a separate
module, such as test_widget.py:

- The test module can be run standalone from the command line.
- The test code can more easily be separated from shipped code.
- There is less temptation to change test code to fit the code it
tests without a good reason.
- Test code should be modified much less frequently than the
code it tests.
- Tested code can be refactored more easily.
- Tests for modules written in C must be in separate modules
anyway, so why not be consistent?
If the testing strategy changes, there is no need to change the source code.
25.3.5. Re-using old test code

Some users will find that they have existing test code that they would like to run from unittest, without converting every old test function to a TestCase subclass.

For this reason, unittest provides a FunctionTestCase class. This subclass of TestCase can be used to wrap an existing test function. Set-up and tear-down functions can also be provided.

Given the following test function:

```python
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

one can create an equivalent test case instance as follows:

```python
testcase = unittest.FunctionTestCase(testSomething)
```

If there are additional set-up and tear-down methods that should be called as part of the test case’s operation, they can also be provided like so:

```python
testcase = unittest.FunctionTestCase(testSomething,
        setUp=makeSomethingDB,
        tearDown=deleteSomethingDB)
```

To make migrating existing test suites easier, unittest supports tests raising AssertionError to indicate test failure. However, it is recommended that you use the explicit TestCase.fail*() and TestCase.assert*() methods instead, as future versions of unittest may treat AssertionError differently.
Note: Even though `FunctionTestCase` can be used to quickly convert an existing test base over to a `unittest`-based system, this approach is not recommended. Taking the time to set up proper `TestCase` subclasses will make future test refactorings infinitely easier.

In some cases, the existing tests may have been written using the `doctest` module. If so, `doctest` provides a `DocTestSuite` class that can automatically build `unittest.TestSuite` instances from the existing `doctest`-based tests.
25.3.6. Skipping tests and expected failures

New in version 2.7.

Unittest supports skipping individual test methods and even whole classes of tests. In addition, it supports marking a test as an “expected failure,” a test that is broken and will fail, but shouldn’t be counted as a failure on a TestResult.

Skipping a test is simply a matter of using the skip() decorator or one of its conditional variants.

Basic skipping looks like this:

```python
class MyTestCase(unittest.TestCase):
    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the lib
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass
```

This is the output of running the example above in verbose mode:

test_format (__main__.MyTestCase) ... skipped 'not supported in
test_nothing (__main__.MyTestCase) ... skipped 'demonstrating s
test_windows_support (__main__.MyTestCase) ... skipped 'require

Ran 3 tests in 0.005s
Classes can be skipped just like methods:

```python
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

`TestCase.setUp()` can also skip the test. This is useful when a resource that needs to be set up is not available.

Expected failures use the `expectedFailure()` decorator.

```python
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

It's easy to roll your own skipping decorators by making a decorator that calls `skip()` on the test when it wants it to be skipped. This decorator skips the test unless the passed object has a certain attribute:

```python
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{} doesn't have {!r}".format(obj, a
```

The following decorators implement test skipping and expected failures:

```python
unittest.skip(reason)
```

Unconditionally skip the decorated test. `reason` should describe why the test is being skipped.
unittest. skipIf(condition, reason)
    Skip the decorated test if condition is true.

unittest. skipUnless(condition, reason)
    Skip the decorated test unless condition is true.

unittest. expectedFailure()
    Mark the test as an expected failure. If the test fails when run, the test is not counted as a failure.

exception unittest.SkipTest(reason)
    This exception is raised to skip a test.

    Usually you can use testCase.skipTest() or one of the skipping decorators instead of raising this directly.

Skipped tests will not have setUp() or tearDown() run around them. Skipped classes will not have setUpClass() or tearDownClass() run.
25.3.7. Classes and functions

This section describes in depth the API of `unittest`.

25.3.7.1. Test cases

```python
class unittest.TestCase(methodName='runTest')
```

Instances of the `TestCase` class represent the smallest testable units in the `unittest` universe. This class is intended to be used as a base class, with specific tests being implemented by concrete subclasses. This class implements the interface needed by the test runner to allow it to drive the test, and methods that the test code can use to check for and report various kinds of failure.

Each instance of `TestCase` will run a single test method: the method named `methodName`. If you remember, we had an earlier example that went something like this:

```python
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_size'))
    suite.addTest(WidgetTestCase('test_resize'))
    return suite
```

Here, we create two instances of `WidgetTestCase`, each of which runs a single test.

`methodName` defaults to `runTest()`.

`TestCase` instances provide three groups of methods: one group used to run the test, another used by the test implementation to check conditions and report failures, and some inquiry methods allowing information about the test itself to be gathered.
Methods in the first group (running the test) are:

**setUp()**
Method called to prepare the test fixture. This is called immediately before calling the test method; other than `AssertionError` or `SkipTest`, any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

**tearDown()**
Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state. Any exception, other than `AssertionError` or `SkipTest`, raised by this method will be considered an additional error rather than a test failure (thus increasing the total number of reported errors). This method will only be called if the `setUp()` succeeds, regardless of the outcome of the test method. The default implementation does nothing.

**setUpClass()**
A class method called before tests in an individual class run. `setUpClass` is called with the class as the only argument and must be decorated as a `classmethod()`:

```python
@classmethod
def setUpClass(cls):
    ...
```

See [Class and Module Fixtures](#) for more details.

*New in version 2.7.*

**tearDownClass()**
A class method called after tests in an individual class have run. `tearDownClass` is called with the class as the only argument and must be decorated as a `classmethod()`:

```python
@classmethod
def tearDownClass(cls):
    ...
```

See Class and Module Fixtures for more details.

*New in version 2.7.*

**run(result=None)**

Run the test, collecting the result into the test result object passed as `result`. If `result` is omitted or `None`, a temporary result object is created (by calling the `defaultTestResult()` method) and used. The result object is not returned to `run()`'s caller.

The same effect may be had by simply calling the `TestCase` instance.

**skipTest(reason)**

Calling this during a test method or `setUp()` skips the current test. See Skipping tests and expected failures for more information.

*New in version 2.7.*

**debug()**

Run the test without collecting the result. This allows exceptions raised by the test to be propagated to the caller, and can be used to support running tests under a debugger.

The `TestCase` class provides several assert methods to check for
and report failures. The following table lists the most commonly used methods (see the tables below for more assert methods):

<table>
<thead>
<tr>
<th>Method</th>
<th>Checks that</th>
<th>New in</th>
</tr>
</thead>
<tbody>
<tr>
<td>assertEqual(a, b)</td>
<td>a == b</td>
<td></td>
</tr>
<tr>
<td>assertNotEqual(a, b)</td>
<td>a != b</td>
<td></td>
</tr>
<tr>
<td>assertTrue(x)</td>
<td>bool(x) is True</td>
<td></td>
</tr>
<tr>
<td>assertFalse(x)</td>
<td>bool(x) is False</td>
<td></td>
</tr>
<tr>
<td>assertIs(a, b)</td>
<td>a is b</td>
<td>2.7</td>
</tr>
<tr>
<td>assertIsNot(a, b)</td>
<td>a is not b</td>
<td>2.7</td>
</tr>
<tr>
<td>assertIsNone(x)</td>
<td>x is None</td>
<td>2.7</td>
</tr>
<tr>
<td>assertIsNotNone(x)</td>
<td>x is not None</td>
<td>2.7</td>
</tr>
<tr>
<td>assertIn(a, b)</td>
<td>a in b</td>
<td>2.7</td>
</tr>
<tr>
<td>assertNotIn(a, b)</td>
<td>a not in b</td>
<td>2.7</td>
</tr>
<tr>
<td>assertIsInstance(a, b)</td>
<td>isinstance(a, b)</td>
<td>2.7</td>
</tr>
<tr>
<td>assertNotIsInstance(a, b)</td>
<td>not isinstance(a, b)</td>
<td>2.7</td>
</tr>
</tbody>
</table>

All the assert methods (except `assertRaises()`, `assertRaisesRegexp()`) accept a `msg` argument that, if specified, is used as the error message on failure (see also `longMessage`).

**assertEqual(first, second, msg=None)**

Test that `first` and `second` are equal. If the values do not compare equal, the test will fail.

In addition, if `first` and `second` are the exact same type and one of list, tuple, dict, set, frozenset or unicode or any type that a subclass registers with `addTypeEqualityFunc()` the type-specific equality function will be called in order to generate a more useful default error message (see also the list of type-specific methods).

*Changed in version 2.7:* Added the automatic calling of type-specific equality function.
assertNotEqual(first, second, msg=None)
    Test that first and second are not equal. If the values do compare equal, the test will fail.

assertTrue(expr, msg=None)
assertFalse(expr, msg=None)
    Test that expr is true (or false).

    Note that this is equivalent to bool(expr) is True and not to expr is True (use assertIs(expr, True) for the latter). This method should also be avoided when more specific methods are available (e.g. assertEqual(a, b) instead of assertTrue(a == b)), because they provide a better error message in case of failure.

assertIs(first, second, msg=None)
assertIsNot(first, second, msg=None)
    Test that first and second evaluate (or don’t evaluate) to the same object.

    New in version 2.7.

assertIsNone(expr, msg=None)
assertIsNotNone(expr, msg=None)
    Test that expr is (or is not) None.

    New in version 2.7.

assertIn(first, second, msg=None)
assertNotIn(first, second, msg=None)
    Test that first is (or is not) in second.

    New in version 2.7.

assertIsInstance(obj, cls, msg=None)
**assertNotIsInstance**(obj, cls, msg=None)

Test that obj is (or is not) an instance of cls (which can be a class or a tuple of classes, as supported by `isinstance()`). To check for the exact type, use `assertIs(type(obj), cls).

*New in version 2.7.*

It is also possible to check that exceptions and warnings are raised using the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Checks that</th>
<th>New in</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertRaises(exc, fun, *args, **kwds)</code></td>
<td><code>fun(*args, **kwds)</code> raises <code>exc</code></td>
<td></td>
</tr>
<tr>
<td><code>assertRaisesRegexp(exc, r, fun, *args, **kwds)</code></td>
<td><code>fun(*args, **kwds)</code> raises <code>exc</code> and the message matches regex <code>r</code></td>
<td>2.7</td>
</tr>
</tbody>
</table>

**assertRaises(exception, callable, *args, **kwds)**

Test that an exception is raised when callable is called with any positional or keyword arguments that are also passed to `assertRaises()`. The test passes if exception is raised, is an error if another exception is raised, or fails if no exception is raised. To catch any of a group of exceptions, a tuple containing the exception classes may be passed as exception.

If only the exception argument is given, returns a context manager so that the code under test can be written inline rather than as a function:

```python
with self.assertRaises(SomeException):
    do_something()
```
The context manager will store the caught exception object in its `exception` attribute. This can be useful if the intention is to perform additional checks on the exception raised:

```python
with self.assertRaises(SomeException) as cm:
    do_something()
the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

*Changed in version 2.7:* Added the ability to use `assertRaises()` as a context manager.

```python
assertRaisesRegexp(exception, regexp, callable, *args, **kwds)
assertRaisesRegexp(exception, regexp)
```

Like `assertRaises()` but also tests that `regexp` matches on the string representation of the raised exception. `regexp` may be a regular expression object or a string containing a regular expression suitable for use by `re.search()`. Examples:

```python
self.assertRaisesRegexp(ValueError, "invalid literal for.*XYZ'\)
```

or:

```python
with self.assertRaisesRegexp(ValueError, 'literal):
    int('XYZ')
```

*New in version 2.7.*

There are also other methods used to perform more specific checks, such as:

<table>
<thead>
<tr>
<th>Method</th>
<th>Checks that</th>
<th>New in</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertAlmostEqual(a, b)</code></td>
<td><code>round(a-b, 7) == 0</code></td>
<td></td>
</tr>
<tr>
<td><code>assertNotAlmostEqual(a, b)</code></td>
<td><code>round(a-b, 7) != 0</code></td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
<td>Version</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td><code>assertGreater(a, b)</code></td>
<td><code>a &gt; b</code></td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertGreaterEqual(a, b)</code></td>
<td><code>a &gt;= b</code></td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertLess(a, b)</code></td>
<td><code>a &lt; b</code></td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertLessEqual(a, b)</code></td>
<td><code>a &lt;= b</code></td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertRegexpMatches(s, r)</code></td>
<td><code>r.search(s)</code></td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertNotRegexpMatches(s, r)</code></td>
<td><code>not r.search(s)</code></td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertItemsEqual(a, b)</code></td>
<td><code>sorted(a) == sorted(b)</code> and works with unhashable objs</td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertDictContainsSubset(a, b)</code></td>
<td>all the key/value pairs in <code>a</code> exist in <code>b</code></td>
<td>2.7</td>
</tr>
</tbody>
</table>

`assertAlmostEqual(first, second, places=7, msg=None, delta=None)`

`assertNotAlmostEqual(first, second, places=7, msg=None, delta=None)`

Test that `first` and `second` are approximately (or not approximately) equal by computing the difference, rounding to the given number of decimal `places` (default 7), and comparing to zero. Note that these methods round the values to the given number of decimal places (i.e. like the `round()` function) and not significant digits.

If `delta` is supplied instead of `places` then the difference between `first` and `second` must be less or equal to (or greater than) `delta`.

Supplying both `delta` and `places` raises a `TypeError`.

*Changed in version 2.7:* `assertAlmostEqual()` automatically considers almost equal objects that compare equal. `assertNotAlmostEqual()` automatically fails if the objects compare equal. Added the `delta` keyword argument.
assertGreater\( (first, second, msg=None) \)
assertGreaterEqual\( (first, second, msg=None) \)
assertLess\( (first, second, msg=None) \)
assertLessEqual\( (first, second, msg=None) \)

Test that \( first \) is respectively \( >, \geq, < \) or \( \leq \) than \( second \) depending on the method name. If not, the test will fail:

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

*New in version 2.7.*

assertRegexpMatches\( (text, regexp, msg=None) \)

Test that a \( regexp \) search matches \( text \). In case of failure, the error message will include the pattern and the \( text \) (or the pattern and the part of \( text \) that unexpectedly matched). \( regexp \) may be a regular expression object or a string containing a regular expression suitable for use by \( re.search() \).

*New in version 2.7.*

assertNotRegexpMatches\( (text, regexp, msg=None) \)

Verifies that a \( regexp \) search does not match \( text \). Fails with an error message including the pattern and the part of \( text \) that matches. \( regexp \) may be a regular expression object or a string containing a regular expression suitable for use by \( re.search() \).

*New in version 2.7.*

assertItemsEqual\( (actual, expected, msg=None) \)

Test that sequence \( expected \) contains the same elements as \( actual \), regardless of their order. When they don’t, an error message listing the differences between the sequences will
be generated.

Duplicate elements are *not* ignored when comparing *actual* and *expected*. It verifies if each element has the same count in both sequences. It is the equivalent of `assertEqual(sorted(expected), sorted(actual))` but it works with sequences of unhashable objects as well.

In Python 3, this method is named `assertCountEqual`.

*New in version 2.7.*

`assertDictContainsSubset(expected, actual, msg=None)`
Tests whether the key/value pairs in dictionary *actual* are a superset of those in *expected*. If not, an error message listing the missing keys and mismatched values is generated.

*New in version 2.7.*

*Deprecated since version 3.2.*

The `assertEqual()` method dispatches the equality check for objects of the same type to different type-specific methods. These methods are already implemented for most of the built-in types, but it’s also possible to register new methods using `addTypeEqualityFunc()`:

`addTypeEqualityFunc(typeobj, function)`
Registers a type-specific method called by `assertEqual()` to check if two objects of exactly the same *typeobj* (not subclasses) compare equal. *function* must take two positional arguments and a third `msg=None` keyword argument just as `assertEqual()` does. It must raise `self.failureException(msg)` when inequality between the first two parameters is detected – possibly providing useful
information and explaining the inequalities in details in the error message.

*New in version 2.7.*

The list of type-specific methods automatically used by `assertEqual()` are summarized in the following table. Note that it's usually not necessary to invoke these methods directly.

<table>
<thead>
<tr>
<th>Method</th>
<th>Used to compare</th>
<th>New in</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertMultiLineEqual(a, b)</code></td>
<td>strings</td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertSequenceEqual(a, b)</code></td>
<td>sequences</td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertListEqual(a, b)</code></td>
<td>lists</td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertTupleEqual(a, b)</code></td>
<td>tuples</td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertSetEqual(a, b)</code></td>
<td>sets or frozensets</td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertDictEqual(a, b)</code></td>
<td>dicts</td>
<td>2.7</td>
</tr>
</tbody>
</table>

`assertMultiLineEqual(first, second, msg=None)`

Test that the multiline string `first` is equal to the string `second`. When not equal a diff of the two strings highlighting the differences will be included in the error message. This method is used by default when comparing strings with `assertEqual()`.

*New in version 2.7.*

`assertSequenceEqual(seq1, seq2, msg=None, seq_type=None)`

Tests that two sequences are equal. If a `seq_type` is supplied, both `seq1` and `seq2` must be instances of `seq_type` or a failure will be raised. If the sequences are different an error message is constructed that shows the difference between the two.

This method is not called directly by `assertEqual()`, but it's used to implement `assertListEqual()` and
assertTupleEqual().

*New in version 2.7.*

**assertListEqual(list1, list2, msg=None)**

**assertTupleEqual(tuple1, tuple2, msg=None)**

Tests that two lists or tuples are equal. If not, an error message is constructed that shows only the differences between the two. An error is also raised if either of the parameters are of the wrong type. These methods are used by default when comparing lists or tuples with `assertEqual()`.

*New in version 2.7.*

**assertSetEqual(set1, set2, msg=None)**

Tests that two sets are equal. If not, an error message is constructed that lists the differences between the sets. This method is used by default when comparing sets or frozensets with `assertEqual()`.

Fails if either of `set1` or `set2` does not have a `set.difference()` method.

*New in version 2.7.*

**assertDictEqual(expected, actual, msg=None)**

Test that two dictionaries are equal. If not, an error message is constructed that shows the differences in the dictionaries. This method will be used by default to compare dictionaries in calls to `assertEqual()`.

*New in version 2.7.*

Finally the **TestCase** provides the following methods and attributes:
fail(msg=None)

Signals a test failure unconditionally, with msg or None for the error message.

failureException

This class attribute gives the exception raised by the test method. If a test framework needs to use a specialized exception, possibly to carry additional information, it must subclass this exception in order to “play fair” with the framework. The initial value of this attribute is AssertionError.

longMessage

If set to True then any explicit failure message you pass in to the assert methods will be appended to the end of the normal failure message. The normal messages contain useful information about the objects involved, for example the message from assertEqual shows you the repr of the two unequal objects. Setting this attribute to True allows you to have a custom error message in addition to the normal one.

This attribute defaults to False, meaning that a custom message passed to an assert method will silence the normal message.

The class setting can be overridden in individual tests by assigning an instance attribute to True or False before calling the assert methods.

New in version 2.7.

maxDiff

This attribute controls the maximum length of diffs output by assert methods that report diffs on failure. It defaults to 80*8 characters. Assert methods affected by this attribute are assertSequenceEqual() (including all the sequence
comparison methods that delegate to it), \texttt{assertDictEqual()} and \texttt{assertMultiLineEqual()}.

Setting \texttt{maxDiff} to \texttt{None} means that there is no maximum length of diffs.

\textit{New in version 2.7.}

Testing frameworks can use the following methods to collect information on the test:

\textbf{countTestCases()}

Return the number of tests represented by this test object. For \texttt{TestCase} instances, this will always be 1.

\textbf{defaultTestResult()}

Return an instance of the test result class that should be used for this test case class (if no other result instance is provided to the \texttt{run()} method).

For \texttt{TestCase} instances, this will always be an instance of \texttt{TestResult}; subclasses of \texttt{TestCase} should override this as necessary.

\textbf{id()}

Return a string identifying the specific test case. This is usually the full name of the test method, including the module and class name.

\textbf{shortDescription()}

Returns a description of the test, or \texttt{None} if no description has been provided. The default implementation of this method returns the first line of the test method's docstring, if available, or \texttt{None}. 
addCleanup(function, *args, **kwargs)
Add a function to be called after tearDown() to cleanup resources used during the test. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into addCleanup() when they are added.

If setUp() fails, meaning that tearDown() is not called, then any cleanup functions added will still be called.

*New in version 2.7.*

doCleanups()
This method is called unconditionally after tearDown(), or after setUp() if setUp() raises an exception.

It is responsible for calling all the cleanup functions added by addCleanup(). If you need cleanup functions to be called prior to tearDown() then you can call doCleanups() yourself.

doCleanups() pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

*New in version 2.7.*

class unittest.FunctionTestCase(testFunc, setUp=None, tearDown=None, description=None)
This class implements the portion of the TestCase interface which allows the test runner to drive the test, but does not provide the methods which test code can use to check and report errors. This is used to create test cases using legacy test code, allowing it to be integrated into a unittest-based test framework.

25.3.7.1.1. Deprecated aliases
For historical reasons, some of the `TestCase` methods had one or more aliases that are now deprecated. The following table lists the correct names along with their deprecated aliases:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Deprecated alias(es)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertEqual()</code></td>
<td><code>failUnlessEqual, assertEquals</code></td>
</tr>
<tr>
<td><code>assertNotEqual()</code></td>
<td><code>failIfEqual</code></td>
</tr>
<tr>
<td><code>assertTrue()</code></td>
<td><code>failUnless, assert_</code></td>
</tr>
<tr>
<td><code>assertFalse()</code></td>
<td><code>failIf</code></td>
</tr>
<tr>
<td><code>assertRaises()</code></td>
<td><code>failUnlessRaises</code></td>
</tr>
<tr>
<td><code>assertAlmostEqual()</code></td>
<td><code>failUnlessAlmostEqual</code></td>
</tr>
<tr>
<td><code>assertNotAlmostEqual()</code></td>
<td><code>failIfAlmostEqual</code></td>
</tr>
</tbody>
</table>

*Deprecated since version 2.7:* the aliases listed in the second column

### 25.3.7.2. Grouping tests

```python
class unittest.TestSuite(tests=())
```

This class represents an aggregation of individual tests cases and test suites. The class presents the interface needed by the test runner to allow it to be run as any other test case. Running a `TestSuite` instance is the same as iterating over the suite, running each test individually.

If `tests` is given, it must be an iterable of individual test cases or other test suites that will be used to build the suite initially. Additional methods are provided to add test cases and suites to the collection later on.

`TestSuite` objects behave much like `TestCase` objects, except they do not actually implement a test. Instead, they are used to
aggregate tests into groups of tests that should be run together. Some additional methods are available to add tests to TestSuite instances:

**addTest(test)**

Add a **TestCase** or **TestSuite** to the suite.

**addTests(tests)**

Add all the tests from an iterable of **TestCase** and **TestSuite** instances to this test suite.

This is equivalent to iterating over *tests*, calling **addTest()** for each element.

**TestSuite** shares the following methods with **TestCase**:

**run(result)**

Run the tests associated with this suite, collecting the result into the test result object passed as *result*. Note that unlike **TestCase.run()**, **TestSuite.run()** requires the result object to be passed in.

**debug()**

Run the tests associated with this suite without collecting the result. This allows exceptions raised by the test to be propagated to the caller and can be used to support running tests under a debugger.

**countTestCases()**

Return the number of tests represented by this test object, including all individual tests and sub-suites.

**__iter__()**

Tests grouped by a **TestSuite** are always accessed by
iteration. Subclasses can lazily provide tests by overriding `__iter__()`. Note that this method maybe called several times on a single suite (for example when counting tests or comparing for equality) so the tests returned must be the same for repeated iterations.

*Changed in version 2.7:* In earlier versions the `TestSuite` accessed tests directly rather than through iteration, so overriding `__iter__()` wasn’t sufficient for providing tests.

In the typical usage of a `TestSuite` object, the `run()` method is invoked by a `TestRunner` rather than by the end-user test harness.

### 25.3.7.3. Loading and running tests

*class* `unittest.TestLoader`

The `TestLoader` class is used to create test suites from classes and modules. Normally, there is no need to create an instance of this class; the `unittest` module provides an instance that can be shared as `unittest.defaultTestLoader`. Using a subclass or instance, however, allows customization of some configurable properties.

`TestLoader` objects have the following methods:

- `loadTestsFromTestCase(testCaseClass)`
  Return a suite of all tests cases contained in the `TestCase`-derived `testCaseClass`.

- `loadTestsFromModule(module)`
  Return a suite of all tests cases contained in the given module. This method searches `module` for classes derived from `TestCase` and creates an instance of the class for each
test method defined for the class.

**Note:** While using a hierarchy of `TestCase`-derived classes can be convenient in sharing fixtures and helper functions, defining test methods on base classes that are not intended to be instantiated directly does not play well with this method. Doing so, however, can be useful when the fixtures are different and defined in subclasses.

If a module provides a `load_tests` function it will be called to load the tests. This allows modules to customize test loading. This is the `load_tests` protocol.

*Changed in version 2.7:* Support for `load_tests` added.

`loadTestsFromName(name, module=None)`

Return a suite of all tests cases given a string specifier.

The specifier `name` is a “dotted name” that may resolve either to a module, a test case class, a test method within a test case class, a `TestSuite` instance, or a callable object which returns a `TestCase` or `TestSuite` instance. These checks are applied in the order listed here; that is, a method on a possible test case class will be picked up as “a test method within a test case class”, rather than “a callable object”.

For example, if you have a module `SampleTests` containing a `TestCase`-derived class `SampleTestCase` with three test methods (`test_one()`, `test_two()`, and `test_three()`), the specifier `'SampleTests.SampleTestCase'` would cause this method to return a suite which will run all three test methods. Using the specifier `'SampleTests.SampleTestCase.test_two'` would cause it to return a test suite which will run only the `test_two()` test method. The specifier can refer to modules
and packages which have not been imported; they will be imported as a side-effect.

The method optionally resolves name relative to the given module.

`loadTestsFromNames(names, module=None)`

Similar to `loadTestsFromName()`, but takes a sequence of names rather than a single name. The return value is a test suite which supports all the tests defined for each name.

`getTestCaseNames(testCaseClass)`

Return a sorted sequence of method names found within `testCaseClass`; this should be a subclass of `TestCase`.

`discover(start_dir, pattern='test*.py', top_level_dir=None)`

Find all the test modules by recursing into subdirectories from the specified start directory, and return a TestSuite object containing them. Only test files that match `pattern` will be loaded. (Using shell style pattern matching.) Only module names that are importable (i.e. are valid Python identifiers) will be loaded.

All test modules must be importable from the top level of the project. If the start directory is not the top level directory then the top level directory must be specified separately.

If importing a module fails, for example due to a syntax error, then this will be recorded as a single error and discovery will continue.

If a test package name (directory with `__init__.py`) matches the pattern then the package will be checked for a `load_tests` function. If this exists then it will be called with `loader, tests, pattern`. 
If `load_tests` exists then discovery does *not* recurse into the package, `load_tests` is responsible for loading all tests in the package.

The pattern is deliberately not stored as a loader attribute so that packages can continue discovery themselves. `top_level_dir` is stored so `load_tests` does not need to pass this argument in to `loader.discover()`.

`start_dir` can be a dotted module name as well as a directory.

*New in version 2.7.*

The following attributes of a `TestLoader` can be configured either by subclassing or assignment on an instance:

**testMethodPrefix**

String giving the prefix of method names which will be interpreted as test methods. The default value is `'test'`.

This affects `getTestCaseNames()` and all the `loadTestsFrom*()` methods.

**sortTestMethodsUsing**

Function to be used to compare method names when sorting them in `getTestCaseNames()` and all the `loadTestsFrom*()` methods. The default value is the built-in `cmp()` function; the attribute can also be set to `None` to disable the sort.

**suiteClass**

Callable object that constructs a test suite from a list of tests. No methods on the resulting object are needed. The default value is the `TestSuite` class.

This affects all the `loadTestsFrom*()` methods.
class unittest.TestResult

This class is used to compile information about which tests have succeeded and which have failed.

A TestResult object stores the results of a set of tests. The TestCase and TestSuite classes ensure that results are properly recorded; test authors do not need to worry about recording the outcome of tests.

Testing frameworks built on top of unittest may want access to the TestResult object generated by running a set of tests for reporting purposes; a TestResult instance is returned by the TestRunner.run() method for this purpose.

TestResult instances have the following attributes that will be of interest when inspecting the results of running a set of tests:

errors

A list containing 2-tuples of TestCase instances and strings holding formatted tracebacks. Each tuple represents a test which raised an unexpected exception.

Changed in version 2.2: Contains formatted tracebacks instead of sys.exc_info() results.

failures

A list containing 2-tuples of TestCase instances and strings holding formatted tracebacks. Each tuple represents a test where a failure was explicitly signalled using the TestCase.assert*() methods.

Changed in version 2.2: Contains formatted tracebacks instead of sys.exc_info() results.

skipped
A list containing 2-tuples of `TestCase` instances and strings holding the reason for skipping the test.

*New in version 2.7.*

**expectedFailures**
A list containing 2-tuples of `TestCase` instances and strings holding formatted tracebacks. Each tuple represents an expected failure of the test case.

**unexpectedSuccesses**
A list containing `TestCase` instances that were marked as expected failures, but succeeded.

**shouldStop**
Set to `True` when the execution of tests should stop by `stop()`.

**testsRun**
The total number of tests run so far.

**buffer**
If set to true, `sys.stdout` and `sys.stderr` will be buffered in between `startTest()` and `stopTest()` being called. Collected output will only be echoed onto the real `sys.stdout` and `sys.stderr` if the test fails or errors. Any output is also attached to the failure / error message.

*New in version 2.7.*

**failfast**
If set to true `stop()` will be called on the first failure or error, halting the test run.

*New in version 2.7.*
**wasSuccessful()**

Return `True` if all tests run so far have passed, otherwise returns `False`.

**stop()**

This method can be called to signal that the set of tests being run should be aborted by setting the `shouldStop` attribute to `True`. **TestRunner** objects should respect this flag and return without running any additional tests.

For example, this feature is used by the **TextTestRunner** class to stop the test framework when the user signals an interrupt from the keyboard. Interactive tools which provide **TestRunner** implementations can use this in a similar manner.

The following methods of the **TestResult** class are used to maintain the internal data structures, and may be extended in subclasses to support additional reporting requirements. This is particularly useful in building tools which support interactive reporting while tests are being run.

**startTest(**`test`**)**

Called when the test case `test` is about to be run.

**stopTest(**`test`**)**

Called after the test case `test` has been executed, regardless of the outcome.

**startTestRun()**

Called once before any tests are executed.

*New in version 2.7.*

**stopTestRun()**
Called once after all tests are executed.

*New in version 2.7.*

**addError** *(test, err)*

Called when the test case *test* raises an unexpected exception. *err* is a tuple of the form returned by `sys.exc_info()`: *(type, value, traceback)*.

The default implementation appends a tuple *(test, formatted_err)* to the instance’s *errors* attribute, where *formatted_err* is a formatted traceback derived from *err*.

**addFailure** *(test, err)*

Called when the test case *test* signals a failure. *err* is a tuple of the form returned by `sys.exc_info()`: *(type, value, traceback)*.

The default implementation appends a tuple *(test, formatted_err)* to the instance’s *failures* attribute, where *formatted_err* is a formatted traceback derived from *err*.

**addSuccess** *(test)*

Called when the test case *test* succeeds.

The default implementation does nothing.

**addSkip** *(test, reason)*

Called when the test case *test* is skipped. *reason* is the reason the test gave for skipping.

The default implementation appends a tuple *(test, reason)* to the instance’s *skipped* attribute.

**addExpectedFailure** *(test, err)*
Called when the test case `test` fails, but was marked with the `expectedFailure()` decorator.

The default implementation appends a tuple `(test, formatted_err)` to the instance's `expectedFailures` attribute, where `formatted_err` is a formatted traceback derived from `err`.

`addUnexpectedSuccess(test)`
Called when the test case `test` was marked with the `expectedFailure()` decorator, but succeeded.

The default implementation appends the test to the instance's `unexpectedSuccesses` attribute.

```python
class unittest.TextTestResult(stream, descriptions, verbosity)
A concrete implementation of TestResult used by the TextTestRunner.

New in version 2.7: This class was previously named _TextTestResult. The old name still exists as an alias but is deprecated.
```

```python
unittest.defaultTestLoader
Instance of the TestLoader class intended to be shared. If no customization of the TestLoader is needed, this instance can be used instead of repeatedly creating new instances.
```

```python
class unittest.TextTestRunner(stream=sys.stderr, descriptions=True, verbosity=1, failfast=False, buffer=False, resultclass=None)
A basic test runner implementation which prints results on standard error. It has a few configurable parameters, but is essentially very simple. Graphical applications which run test suites should provide alternate implementations.
```
_makeResult()

This method returns the instance of TestResult used by run(). It is not intended to be called directly, but can be overridden in subclasses to provide a custom TestResult.

_makeResult() instantiates the class or callable passed in the TextTestRunner constructor as the resultclass argument. It defaults to TextTestResult if no resultclass is provided. The result class is instantiated with the following arguments:

stream, descriptions, verbosity

unittest.main([module[, defaultTest[, argv[, testRunner[, testLoader[, exit[, verbosity[, failfast[, catchbreak[, buffer]]]]]]]]]]])

A command-line program that loads a set of tests from module and runs them; this is primarily for making test modules conveniently executable. The simplest use for this function is to include the following line at the end of a test script:

if __name__ == '__main__':
    unittest.main()

You can run tests with more detailed information by passing in the verbosity argument:

if __name__ == '__main__':
    unittest.main(verbosity=2)

The defaultTest argument is the name of the test to run if no test names are specified via argv. If not specified or None and no test names are provided via argv, all tests found in module are run.

The argv argument can be a list of options passed to the program, with the first element being the program name. If not specified or None, the values of sys.argv are used.
The *testRunner* argument can either be a test runner class or an already created instance of it. By default *main* calls *sys.exit()* with an exit code indicating success or failure of the tests run.

The *testLoader* argument has to be a *TestLoader* instance, and defaults to *defaultTestLoader*.

*main* supports being used from the interactive interpreter by passing in the argument *exit=False*. This displays the result on standard output without calling *sys.exit()*:

```python
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

The *failfast*, *catchbreak* and *buffer* parameters have the same effect as the same-name command-line options.

Calling *main* actually returns an instance of the *TestProgram* class. This stores the result of the tests run as the *result* attribute.

*Changed in version 2.7*: The *exit*, *verbosity*, *failfast*, *catchbreak* and *buffer* parameters were added.

### 25.3.7.3.1. load_tests Protocol

*New in version 2.7.*

Modules or packages can customize how tests are loaded from them during normal test runs or test discovery by implementing a function called *load_tests*.

If a test module defines *load_tests* it will be called by *TestLoader.loadTestsFromModule()* with the following arguments:
load_tests(loader, standard_tests, None)

It should return a TestSuite.

loader is the instance of TestLoader doing the loading. standard_tests are the tests that would be loaded by default from the module. It is common for test modules to only want to add or remove tests from the standard set of tests. The third argument is used when loading packages as part of test discovery.

A typical load_tests function that loads tests from a specific set of TestCase classes may look like:

test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite

If discovery is started, either from the command line or by calling TestLoader.discover(), with a pattern that matches a package name then the package __init__.py will be checked for load_tests.

**Note:** The default pattern is 'test*.py'. This matches all Python files that start with 'test' but won't match any test directories.

A pattern like 'test*' will match test packages as well as modules.

If the package __init__.py defines load_tests then it will be called and discovery not continued into the package. load_tests is called with the following arguments:

load_tests(loader, standard_tests, pattern)
This should return a `TestSuite` representing all the tests from the package. (`standard_tests` will only contain tests collected from `__init__.py`.)

Because the pattern is passed into `load_tests` the package is free to continue (and potentially modify) test discovery. A ‘do nothing’ `load_tests` function for a test package would look like:

```python
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir, pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```
25.3.8. Class and Module Fixtures

Class and module level fixtures are implemented in TestSuite. When the test suite encounters a test from a new class then tearDownClass() from the previous class (if there is one) is called, followed by setUpClass() from the new class.

Similarly if a test is from a different module from the previous test then tearDownModule from the previous module is run, followed by setUpModule from the new module.

After all the tests have run the final tearDownClass and tearDownModule are run.

Note that shared fixtures do not play well with [potential] features like test parallelization and they break test isolation. They should be used with care.

The default ordering of tests created by the unittest test loaders is to group all tests from the same modules and classes together. This will lead to setUpClass / setUpModule (etc) being called exactly once per class and module. If you randomize the order, so that tests from different modules and classes are adjacent to each other, then these shared fixture functions may be called multiple times in a single test run.

Shared fixtures are not intended to work with suites with non-standard ordering. A BaseTestSuite still exists for frameworks that don’t want to support shared fixtures.

If there are any exceptions raised during one of the shared fixture functions the test is reported as an error. Because there is no corresponding test instance an _ErrorHolder object (that has the
same interface as a `TestCase`) is created to represent the error. If you are just using the standard unittest test runner then this detail doesn’t matter, but if you are a framework author it may be relevant.

### 25.3.8.1. setUpClass and tearDownClass

These must be implemented as class methods:

```python
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

If you want the `setUpClass` and `tearDownClass` on base classes called then you must call up to them yourself. The implementations in `TestCase` are empty.

If an exception is raised during a `setUpClass` then the tests in the class are not run and the `tearDownClass` is not run. Skipped classes will not have `setUpClass` or `tearDownClass` run. If the exception is a `SkipTest` exception then the class will be reported as having been skipped instead of as an error.

### 25.3.8.2. setUpModule and tearDownModule

These should be implemented as functions:

```python
def setUpModule():
    createConnection()

def tearDownModule():
```
If an exception is raised in a `setUpModule` then none of the tests in the module will be run and the `tearDownModule` will not be run. If the exception is a `SkipTest` exception then the module will be reported as having been skipped instead of as an error.
25.3.9. Signal Handling

The `-c/--catch` command-line option to `unittest`, along with the `catchbreak` parameter to `unittest.main()`, provide more friendly handling of control-C during a test run. With catch break behavior enabled control-C will allow the currently running test to complete, and the test run will then end and report all the results so far. A second control-c will raise a `KeyboardInterrupt` in the usual way.

The control-c handling signal handler attempts to remain compatible with code or tests that install their own `signal.SIGINT` handler. If the `unittest` handler is called but isn’t the installed `signal.SIGINT` handler, i.e. it has been replaced by the system under test and delegated to, then it calls the default handler. This will normally be the expected behavior by code that replaces an installed handler and delegates to it. For individual tests that need `unittest` control-c handling disabled the `removeHandler()` decorator can be used.

There are a few utility functions for framework authors to enable control-c handling functionality within test frameworks.

`unittest.installHandler()`

Install the control-c handler. When a `signal.SIGINT` is received (usually in response to the user pressing control-c) all registered results have `stop()` called.

*New in version 2.7.*

`unittest.registerResult(result)`

Register a `TestResult` object for control-c handling. Registering a result stores a weak reference to it, so it doesn’t prevent the result from being garbage collected.
Registering a `TestResult` object has no side-effects if control-c handling is not enabled, so test frameworks can unconditionally register all results they create independently of whether or not handling is enabled.

*New in version 2.7.*

`unittest.removeResult(result)`

Remove a registered result. Once a result has been removed then `stop()` will no longer be called on that result object in response to a control-c.

*New in version 2.7.*

`unittest.removeHandler(function=None)`

When called without arguments this function removes the control-c handler if it has been installed. This function can also be used as a test decorator to temporarily remove the handler whilst the test is being executed:

```python
@unittest.removeHandler
def test_signal_handling(self):
    ...
```

*New in version 2.7.*
25.4. 2to3 - Automated Python 2 to 3 code translation

2to3 is a Python program that reads Python 2.x source code and applies a series of fixers to transform it into valid Python 3.x code. The standard library contains a rich set of fixers that will handle almost all code. 2to3 supporting library lib2to3 is, however, a flexible and generic library, so it is possible to write your own fixers for 2to3. lib2to3 could also be adapted to custom applications in which Python code needs to be edited automatically.
25.4.1. Using 2to3

2to3 will usually be installed with the Python interpreter as a script. It is also located in the `Tools/scripts` directory of the Python root.

2to3’s basic arguments are a list of files or directories to transform. The directories are recursively traversed for Python sources.

Here is a sample Python 2.x source file, `example.py`:

```python
def greet(name):
    print "Hello, {0}".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

It can be converted to Python 3.x code via 2to3 on the command line:

```
$ 2to3 example.py
```

A diff against the original source file is printed. 2to3 can also write the needed modifications right back to the source file. (A backup of the original file is made unless `-n` is also given.) Writing the changes back is enabled with the `-w` flag:

```
$ 2to3 -w example.py
```

After transformation, `example.py` looks like this:

```python
def greet(name):
    print("Hello, {0}".format(name))
print("What's your name?")
name = input()
greet(name)
```
Comments and exact indentation are preserved throughout the translation process.

By default, 2to3 runs a set of predefined fixers. The -1 flag lists all available fixers. An explicit set of fixers to run can be given with -f. Likewise the -x explicitly disables a fixer. The following example runs only the imports and has_key fixers:

```
$ 2to3 -f imports -f has_key example.py
```

This command runs every fixer except the apply fixer:

```
$ 2to3 -x apply example.py
```

Some fixers are explicit, meaning they aren’t run by default and must be listed on the command line to be run. Here, in addition to the default fixers, the idioms fixer is run:

```
$ 2to3 -f all -f idioms example.py
```

Notice how passing all enables all default fixers.

Sometimes 2to3 will find a place in your source code that needs to be changed, but 2to3 cannot fix automatically. In this case, 2to3 will print a warning beneath the diff for a file. You should address the warning in order to have compliant 3.x code.

2to3 can also refactor doctests. To enable this mode, use the -d flag. Note that only doctests will be refactored. This also doesn’t require the module to be valid Python. For example, doctest like examples in a reST document could also be refactored with this option.

The -v option enables output of more information on the translation process.
Since some print statements can be parsed as function calls or statements, 2to3 cannot always read files containing the print function. When 2to3 detects the presence of the \texttt{from \_\_future\_ import print\_function} compiler directive, it modifies its internal grammar to interpret \texttt{print()} as a function. This change can also be enabled manually with the \texttt{-p} flag. Use \texttt{-p} to run fixers on code that already has had its print statements converted.

The \texttt{-o} or \texttt{--output-dir} option allows specification of an alternate directory for processed output files to be written to. The \texttt{-n} flag is required when using this as backup files do not make sense when not overwriting the input files.

\textit{New in version 2.7.3:} The \texttt{-o} option was added.

The \texttt{-W} or \texttt{--write-unchanged-files} flag tells 2to3 to always write output files even if no changes were required to the file. This is most useful with \texttt{-o} so that an entire Python source tree is copied with translation from one directory to another. This option implies the \texttt{-w} flag as it would not make sense otherwise.

\textit{New in version 2.7.3:} The \texttt{-w} flag was added.

The \texttt{--add-suffix} option specifies a string to append to all output filenames. The \texttt{-n} flag is required when specifying this as backups are not necessary when writing to different filenames. Example:

\begin{verbatim}
$ 2to3 -n -W --add-suffix=3 example.py
\end{verbatim}

Will cause a converted file named \texttt{example.py3} to be written.

\textit{New in version 2.7.3:} The \texttt{--add-suffix} option was added.

To translate an entire project from one directory tree to another use:
$ 2to3 --output-dir=python3-version/mycode -W -n python2-version/mycode
25.4.2. Fixers

Each step of transforming code is encapsulated in a fixer. The command `2to3 -l` lists them. As documented above, each can be turned on and off individually. They are described here in more detail.

**apply**

Removes usage of `apply()`. For example `apply(function, *args, **kwargs)` is converted to `function(*args, **kwargs)`.

**asserts**

Replaces deprecated `unittest` method names with the correct ones.

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>failUnlessEqual(a, b)</td>
<td><code>assertEqual(a, b)</code></td>
</tr>
<tr>
<td>assertEquals(a, b)</td>
<td><code>assertEqual(a, b)</code></td>
</tr>
<tr>
<td>failIfEqual(a, b)</td>
<td><code>assertNotEqual(a, b)</code></td>
</tr>
<tr>
<td>assertNotEquals(a, b)</td>
<td><code>assertNotEqual(a, b)</code></td>
</tr>
<tr>
<td>failUnless(a)</td>
<td><code>assertTrue(a)</code></td>
</tr>
<tr>
<td>assert_(a)</td>
<td><code>assertTrue(a)</code></td>
</tr>
<tr>
<td>failIf(a)</td>
<td><code>assertFalse(a)</code></td>
</tr>
<tr>
<td>failUnlessRaises(exc, cal)</td>
<td><code>assertRaises(exc, cal)</code></td>
</tr>
<tr>
<td>failUnlessAlmostEqual(a, b)</td>
<td><code>assertAlmostEqual(a, b)</code></td>
</tr>
<tr>
<td>assertAlmostEquals(a, b)</td>
<td><code>assertAlmostEqual(a, b)</code></td>
</tr>
<tr>
<td>failIfAlmostEqual(a, b)</td>
<td><code>assertNotAlmostEqual(a, b)</code></td>
</tr>
<tr>
<td>assertNotAlmostEquals(a, b)</td>
<td><code>assertNotAlmostEqual(a, b)</code></td>
</tr>
</tbody>
</table>

**basestring**

Converts `basestring` to `str`.

**buffer**
Converts buffer to memoryview. This fixer is optional because the memoryview API is similar but not exactly the same as that of buffer.

callable
Converting callable(x) to isinstance(x, collections.Callable), adding an import to collections if needed. Note callable(x) has returned in Python 3.2, so if you do not intend to support Python 3.1, you can disable this fixer.

dict
Fixes dictionary iteration methods. dict.iteritems() is converted to dict.items(), dict.iterkeys() to dict.keys(), and dict.itervalues() to dict.values(). Similarly, dict.viewitems(), dict.viewkeys() and dict.viewvalues() are converted respectively to dict.items(), dict.keys() and dict.values(). It also wraps existing usages of dict.items(), dict.keys(), and dict.values() in a call to list.

except
Converting except X, T to except X as T.

exec
Converting the exec statement to the exec() function.

execfile
Removes usage of execfile(). The argument to execfile() is wrapped in calls to open(), compile(), and exec().

exitfunc
Changes assignment of sys.exitfunc to use of the atexit module.

filter
Wraps \texttt{filter()} usage in a \texttt{list} call.

\textbf{funcattrs}

Fixes function attributes that have been renamed. For example, \texttt{my\_function.func\_closure} is converted to \texttt{my\_function.\_closure\_}.

\textbf{future}

Removes \texttt{from \_\_future\_ import new\_feature} statements.

\textbf{getcwdu}

Renames \texttt{os.getcwdu()} to \texttt{os.getcwd()}.

\textbf{has\_key}

Changes \texttt{dict.has\_key(key)} to \texttt{key in dict}.

\textbf{idioms}

This optional fixer performs several transformations that make Python code more idiomatic. Type comparisons like \texttt{type(x) is SomeClass} and \texttt{type(x) == SomeClass} are converted to \texttt{isinstance(x, SomeClass)}. \texttt{while 1} becomes \texttt{while True}. This fixer also tries to make use of \texttt{sorted()} in appropriate places. For example, this block

\begin{verbatim}
L = list(some_iterable)
L.sort()
\end{verbatim}

is changed to

\begin{verbatim}
L = sorted(some_iterable)
\end{verbatim}

\textbf{import}

Detects sibling imports and converts them to relative imports.

\textbf{imports}

Handles module renames in the standard library.
imports2
Handles other modules renames in the standard library. It is separate from the imports fixer only because of technical limitations.

input
Converts input(prompt) to eval(input(prompt)).

intern
Converts intern() to sys.intern().

isinstance
Fixes duplicate types in the second argument of isinstance(). For example, isinstance(x, (int, int)) is converted to isinstance(x, (int)).

itertools_imports
Removes imports of itertools.ifilter(), itertools.izip(), and itertools.imap(). Imports of itertools.ifilterfalse() are also changed to itertools.filterfalse().

itertools
Changes usage of itertools.ifilter(), itertools.izip(), and itertools.imap() to their built-in equivalents. itertools.ifilterfalse() is changed to itertools.filterfalse().

long
Renames long to int.

map
Wraps map() in a list call. It also changes map(None, x) to list(x). Using from future_builtins import map disables this fixer.
**metaclass**

Converts the old metaclass syntax (`__metaclass__ = Meta` in the class body) to the new `(class X(metaclass=Meta))`.

**methodattrs**

Fixes old method attribute names. For example, `meth.im_func` is converted to `meth.__func__`.

**ne**

Converts the old not-equal syntax, `<>`, to `!=`.

**next**

Converts the use of iterator's `next()` methods to the `next()` function. It also renames `next()` methods to `__next__()`.

**nonzero**

Renames `__nonzero__()` to `__bool__()`.

**numliterals**

Converts octal literals into the new syntax.

**paren**

Add extra parenthesis where they are required in list comprehensions. For example, `[x for x in 1, 2]` becomes `[x for x in (1, 2)]`.

**print**

Converts the `print` statement to the `print()` function.

**raise**

Converts `raise E, V` to `raise E(V)`, and `raise E, V, T` to `raise E(V).with_traceback(T)`. If `E` is a tuple, the translation will be incorrect because substituting tuples for exceptions has been removed in Python 3.

**raw_input**
Converts \texttt{raw\_input()} to \texttt{input()}.

\textbf{reduce}

Handles the move of \texttt{reduce()} to \texttt{functools.reduce()}.

\textbf{renames}

Changes \texttt{sys.maxint} to \texttt{sys.maxsize}.

\textbf{repr}

Replaces backtick \texttt{repr} with the \texttt{repr()} function.

\textbf{set\_literal}

Replaces use of the \texttt{set} constructor with set literals. This fixer is optional.

\textbf{standarderror}

Renames \texttt{StandardError} to \texttt{Exception}.

\textbf{sys\_exc}

Changes the deprecated \texttt{sys.exc\_value, sys.exc\_type, sys.exc\_traceback} to use \texttt{sys.exc\_info()}.

\textbf{throw}

Fixes the API change in generator's \texttt{throw()} method.

\textbf{tuple\_params}

Removes implicit tuple parameter unpacking. This fixer inserts temporary variables.

\textbf{types}

Fixes code broken from the removal of some members in the \texttt{types} module.

\textbf{unicode}

Renames \texttt{unicode} to \texttt{str}.
urllib
Handles the rename of urllib and urllib2 to the urllib package.

ws_comma
Removes excess whitespace from comma separated items. This fixer is optional.

xrange
Renames xrange() to range() and wraps existing range() calls with list.

xreadlines
Changes for x in file.xreadlines() to for x in file.

zip
Wraps zip() usage in a list call. This is disabled when from future_builtins import zip appears.
25.4.3. *lib2to3* - 2to3's library

**Note:** The *lib2to3* API should be considered unstable and may change drastically in the future.
25.5. **test** — Regression tests package for Python

**Note:** The `test` package is meant for internal use by Python only. It is documented for the benefit of the core developers of Python. Any use of this package outside of Python’s standard library is discouraged as code mentioned here can change or be removed without notice between releases of Python.

The `test` package contains all regression tests for Python as well as the modules `test.test_support` and `test.regrtest`. `test.test_support` is used to enhance your tests while `test.regrtest` drives the testing suite.

Each module in the `test` package whose name starts with `test_` is a testing suite for a specific module or feature. All new tests should be written using the `unittest` or `doctest` module. Some older tests are written using a “traditional” testing style that compares output printed to `sys.stdout`; this style of test is considered deprecated.

**See also:**

- **Module unittest**
  - Writing PyUnit regression tests.
- **Module doctest**
  - Tests embedded in documentation strings.
25.5.1. Writing Unit Tests for the test package

It is preferred that tests that use the `unittest` module follow a few guidelines. One is to name the test module by starting it with `test_` and end it with the name of the module being tested. The test methods in the test module should start with `test_` and end with a description of what the method is testing. This is needed so that the methods are recognized by the test driver as test methods. Also, no documentation string for the method should be included. A comment (such as `# Tests function returns only True or False`) should be used to provide documentation for test methods. This is done because documentation strings get printed out if they exist and thus what test is being run is not stated.

A basic boilerplate is often used:

```python
import unittest
from test import test_support

class MyTestCase1(unittest.TestCase):
    # Only use setUp() and tearDown() if necessary

    def setUp(self):
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...

    def test_feature_two(self):
        # Test feature two.
        ... testing code ...
```
This boilerplate code allows the testing suite to be run by `test.regrtest` as well as on its own as a script.

The goal for regression testing is to try to break code. This leads to a few guidelines to be followed:

- The testing suite should exercise all classes, functions, and constants. This includes not just the external API that is to be presented to the outside world but also “private” code.

- Whitebox testing (examining the code being tested when the tests are being written) is preferred. Blackbox testing (testing only the published user interface) is not complete enough to make sure all boundary and edge cases are tested.

- Make sure all possible values are tested including invalid ones. This makes sure that not only all valid values are acceptable but also that improper values are handled correctly.

- Exhaust as many code paths as possible. Test where branching occurs and thus tailor input to make sure as many different paths through the code are taken.
- Add an explicit test for any bugs discovered for the tested code. This will make sure that the error does not crop up again if the code is changed in the future.

- Make sure to clean up after your tests (such as close and remove all temporary files).

- If a test is dependent on a specific condition of the operating system then verify the condition already exists before attempting the test.

- Import as few modules as possible and do it as soon as possible. This minimizes external dependencies of tests and also minimizes possible anomalous behavior from side-effects of importing a module.

- Try to maximize code reuse. On occasion, tests will vary by something as small as what type of input is used. Minimize code duplication by subclassing a basic test class with a class that specifies the input:

```python
class TestFuncAcceptsSequences(unittest.TestCase):
    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequences):
    arg = [1, 2, 3]

class AcceptStrings(TestFuncAcceptsSequences):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequences):
    arg = (1, 2, 3)
```

See also:
Test Driven Development
A book by Kent Beck on writing tests before code.
25.5.2. Running tests using the command-line interface

The test.regrtest module can be run as a script to drive Python's regression test suite, thanks to the -m option: python -m test.regrtest. Running the script by itself automatically starts running all regression tests in the test package. It does this by finding all modules in the package whose name starts with test_, importing them, and executing the function test_main() if present. The names of tests to execute may also be passed to the script. Specifying a single regression test (python -m test.regrtest test_spam) will minimize output and only print whether the test passed or failed and thus minimize output.

Running test.regrtest directly allows what resources are available for tests to use to be set. You do this by using the -u command-line option. Specifying all as the value for the -u option enables all possible resources: python -m test.regrtest -u all. If all but one resource is desired (a more common case), a comma-separated list of resources that are not desired may be listed after all. The command python -m test.regrtest -u all,-audio,-largefile will run test.regrtest with all resources except the audio and largefile resources. For a list of all resources and more command-line options, run python -m test.regrtest -h.

Some other ways to execute the regression tests depend on what platform the tests are being executed on. On Unix, you can run make test at the top-level directory where Python was built. On Windows, executing rt.bat from your PBuild directory will run all regression tests.
25.6. `test.test_support` — Utility functions for tests

**Note:** The `test.test_support` module has been renamed to `test.support` in Python 3.x.

The `test.test_support` module provides support for Python's regression tests.

This module defines the following exceptions:

```python
exception test.test_support.TestFailed
    Exception to be raised when a test fails. This is deprecated in favor of unittest-based tests and unittest.TestCase's assertion methods.
```

```python
exception test.test_support.ResourceDenied
    Subclass of unittest.SkipTest. Raised when a resource (such as a network connection) is not available. Raised by the requires() function.
```

The `test.test_support` module defines the following constants:

```python
test.test_support.verbose
    True when verbose output is enabled. Should be checked when more detailed information is desired about a running test. `verbose` is set by test.regrtest.
```

```python
test.test_support.have_unicode
    True when Unicode support is available.
```

```python
test.test_support.is_jython
    True if the running interpreter is Jython.
```
test.test_support.TESTFN

Set to a name that is safe to use as the name of a temporary file. Any temporary file that is created should be closed and unlinked (removed).

The test.test_support module defines the following functions:

test.test_support.forget(module_name)

Remove the module named module_name from sys.modules and delete any byte-compiled files of the module.

test.test_support.is_resource_enabled(resource)

Return True if resource is enabled and available. The list of available resources is only set when test.regrtest is executing the tests.

test.test_support.requires(resource[, msg])

Raise ResourceDenied if resource is not available. msg is the argument to ResourceDenied if it is raised. Always returns True if called by a function whose __name__ is '__main__'. Used when tests are executed by test.regrtest.

test.test_support.findfile(filename)

Return the path to the file named filename. If no match is found filename is returned. This does not equal a failure since it could be the path to the file.

test.test_support.run_unittest(*classes)

Execute unittest.TestCase subclasses passed to the function. The function scans the classes for methods starting with the prefix test__ and executes the tests individually.

It is also legal to pass strings as parameters; these should be keys in sys.modules. Each associated module will be scanned by
unittest.TestLoader.loadTestsFromModule(). This is usually seen in the following `test_main()` function:

```python
def test_main():
    test_support.run_unittest(__name__)
```

This will run all tests defined in the named module.

test.test_support.check_warnings(*filters, quiet=True)

A convenience wrapper for `warnings.catch_warnings()` that makes it easier to test that a warning was correctly raised. It is approximately equivalent to calling

```python
warnings.catch_warnings(record=True)
```

with `warningssimplefilter()` set to `always` and with the option to automatically validate the results that are recorded.

`check_warnings` accepts 2-tuples of the form ("message\sregexp", `WarningCategory`) as positional arguments. If one or more `filters` are provided, or if the optional keyword argument `quiet` is `False`, it checks to make sure the warnings are as expected: each specified filter must match at least one of the warnings raised by the enclosed code or the test fails, and if any warnings are raised that do not match any of the specified filters the test fails. To disable the first of these checks, set `quiet` to `True`.

If no arguments are specified, it defaults to:

```python
check_warnings(("", Warning), quiet=True)
```

In this case all warnings are caught and no errors are raised.

On entry to the context manager, a `WarningRecorder` instance is returned. The underlying warnings list from `catch_warnings()` is available via the recorder object’s `warnings` attribute. As a convenience, the attributes of the object representing the most
recent warning can also be accessed directly through the recorder object (see example below). If no warning has been raised, then any of the attributes that would otherwise be expected on an object representing a warning will return None.

The recorder object also has a reset() method, which clears the warnings list.

The context manager is designed to be used like this:

```python
with check_warnings("assertion is always true", SyntaxWarning("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

In this case if either warning was not raised, or some other warning was raised, check_warnings() would raise an error.

When a test needs to look more deeply into the warnings, rather than just checking whether or not they occurred, code like this can be used:

```python
with check_warnings(quiet=True) as w:
    warnings.warn("foo")
    assert str(w.args[0]) == "foo"
    warnings.warn("bar")
    assert str(w.args[0]) == "bar"
    assert str(w.warnings[0].args[0]) == "foo"
    assert str(w.warnings[1].args[0]) == "bar"
    w.reset()
    assert len(w.warnings) == 0
```

Here all warnings will be caught, and the test code tests the captured warnings directly.

*New in version 2.6.*

*Changed in version 2.7:* New optional arguments filters and quiet.
test.test_support.**check_py3k_warnings(***filters*, *quiet=False*)

Similar to **check_warnings()**, but for Python 3 compatibility warnings. If `sys.py3kwarning == 1`, it checks if the warning is effectively raised. If `sys.py3kwarning == 0`, it checks that no warning is raised. It accepts 2-tuples of the form ("message regexp", `WarningCategory`) as positional arguments. When the optional keyword argument `quiet` is `True`, it does not fail if a filter catches nothing. Without arguments, it defaults to:

```
check_py3k_warnings(("", `DeprecationWarning`), quiet=False)
```

*New in version 2.7.*

**test.test_support.captured_stdout**

This is a context manager that runs the `with` statement body using a `StringIO.StringIO` object as `sys.stdout`. That object can be retrieved using the `as` clause of the `with` statement.

Example use:

```
with captured_stdout() as s:
    print "hello"
assert s.getvalue() == "hello\n"
```

*New in version 2.6.*

**test.test_support.import_module**(name, deprecated=False)

This function imports and returns the named module. Unlike a normal import, this function raises `unittest.SkipTest` if the module cannot be imported.

Module and package deprecation messages are suppressed during this import if `deprecated` is `True`.

*New in version 2.7.*
test.test_support.import_fresh_module(name, fresh=(), blocked=(), deprecated=False)

This function imports and returns a fresh copy of the named Python module by removing the named module from `sys.modules` before doing the import. Note that unlike `reload()`, the original module is not affected by this operation.

*fresh* is an iterable of additional module names that are also removed from the `sys.modules` cache before doing the import.

*blocked* is an iterable of module names that are replaced with `0` in the module cache during the import to ensure that attempts to import them raise `ImportError`.

The named module and any modules named in the *fresh* and *blocked* parameters are saved before starting the import and then reinserted into `sys.modules` when the fresh import is complete.

Module and package deprecation messages are suppressed during this import if *deprecated* is `True`.

This function will raise `unittest.SkipTest` if the named module cannot be imported.

Example use:

```python
# Get copies of the warnings module for testing without
# affecting the version being used by the rest of the test suite.
# One copy uses the C implementation, the other is forced to
# the pure Python fallback implementation
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

*New in version 2.7.*

The `test.test_support` module defines the following classes:
class test.test_support.TransientResource(exc[, **kwargs])

Instances are a context manager that raises ResourceDenied if the specified exception type is raised. Any keyword arguments are treated as attribute/value pairs to be compared against any exception raised within the with statement. Only if all pairs match properly against attributes on the exception is ResourceDenied raised.

New in version 2.6.

class test.test_support.EnvironmentVarGuard

Class used to temporarily set or unset environment variables. Instances can be used as a context manager and have a complete dictionary interface for querying/modifying the underlying os.environ. After exit from the context manager all changes to environment variables done through this instance will be rolled back.

New in version 2.6.

Changed in version 2.7: Added dictionary interface.

EnvironmentVarGuard.set(envvar, value)

Temporarily set the environment variable envvar to the value of value.

EnvironmentVarGuard.unset(envvar)

Temporarily unset the environment variable envvar.

class test.test_support.WarningsRecorder

Class used to record warnings for unit tests. See documentation of check_warnings() above for more details.

New in version 2.6.
These libraries help you with Python development: the debugger enables you to step through code, analyze stack frames and set breakpoints etc., and the profilers run code and give you a detailed breakdown of execution times, allowing you to identify bottlenecks in your programs.

- **26.1. bdb** — Debugger framework
- **26.2. pdb** — The Python Debugger
- **26.3. Debugger Commands**
- **26.4. The Python Profilers**
  - **26.4.1. Introduction to the profilers**
  - **26.4.2. Instant User’s Manual**
  - **26.4.3. profile and cProfile Module Reference**
  - **26.4.4. The stats Class**
  - **26.4.5. What Is Deterministic Profiling?**
  - **26.4.6. Limitations**
  - **26.4.7. Calibration**
  - **26.4.8. Using a custom timer**
- **26.5. hotshot** — High performance logging profiler
  - **26.5.1. Profile Objects**
  - **26.5.2. Using hotshot data**
  - **26.5.3. Example Usage**
- **26.6. timeit** — Measure execution time of small code snippets
  - **26.6.1. Basic Examples**
  - **26.6.2. Python Interface**
  - **26.6.3. Command-Line Interface**
  - **26.6.4. Examples**
- **26.7. trace** — Trace or track Python statement execution
  - **26.7.1. Command-Line Usage**
    - **26.7.1.1. Main options**
    - **26.7.1.2. Modifiers**
- 26.7.1.3. Filters
- 26.7.2. Programmatic Interface
The `bdb` module handles basic debugger functions, like setting breakpoints or managing execution via the debugger.

The following exception is defined:

```
exception bdb.BdbQuit
```

Exception raised by the `Bdb` class for quitting the debugger.

The `bdb` module also defines two classes:

```
class bdb.Breakpoint(self, file, line, temporary=0, cond=None, funcname=None)
```

This class implements temporary breakpoints, ignore counts, disabling and (re-)enabling, and conditionals.

Breakpoints are indexed by number through a list called `bpbynumber` and by `(file, line)` pairs through `bplist`. The former points to a single instance of class `Breakpoint`. The latter points to a list of such instances since there may be more than one breakpoint per line.

When creating a breakpoint, its associated filename should be in canonical form. If a `funcname` is defined, a breakpoint hit will be counted when the first line of that function is executed. A conditional breakpoint always counts a hit.

`Breakpoint` instances have the following methods:

```
deleteMe()
```

Delete the breakpoint from the list associated to a file/line. If it
is the last breakpoint in that position, it also deletes the entry for the file/line.

**enable()**

Mark the breakpoint as enabled.

**disable()**

Mark the breakpoint as disabled.

**pprint([out])**

Print all the information about the breakpoint:

- The breakpoint number.
- If it is temporary or not.
- Its file, line position.
- The condition that causes a break.
- If it must be ignored the next N times.
- The breakpoint hit count.

```python
class bdb.Bdb(skip=None)
```

The `Bdb` class acts as a generic Python debugger base class.

This class takes care of the details of the trace facility; a derived class should implement user interaction. The standard debugger class (`pdb.Pdb`) is an example.

The `skip` argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. Whether a frame is considered to originate in a certain module is determined by the `__name__` in the frame globals.

*New in version 2.7:* The `skip` argument.

The following methods of `Bdb` normally don’t need to be
canonic(filename)

Auxiliary method for getting a filename in a canonical form, that is, as a case-normalized (on case-insensitive filesystems) absolute path, stripped of surrounding angle brackets.

reset()

Set the botframe, stopframe, returnframe and quitting attributes with values ready to start debugging.

trace_dispatch(frame, event, arg)

This function is installed as the trace function of debugged frames. Its return value is the new trace function (in most cases, that is, itself).

The default implementation decides how to dispatch a frame, depending on the type of event (passed as a string) that is about to be executed. event can be one of the following:

- "line": A new line of code is going to be executed.
- "call": A function is about to be called, or another code block entered.
- "return": A function or other code block is about to return.
- "exception": An exception has occurred.
- "c_call": A C function is about to be called.
- "c_return": A C function has returned.
- "c_exception": A C function has raised an exception.

For the Python events, specialized functions (see below) are called. For the C events, no action is taken.

The arg parameter depends on the previous event.
See the documentation for `sys.settrace()` for more information on the trace function. For more information on code and frame objects, refer to The standard type hierarchy.

**dispatch_line**(frame)
If the debugger should stop on the current line, invoke the `user_line()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `Bdb.quitting` flag is set (which can be set from `user_line()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

**dispatch_call**(frame, arg)
If the debugger should stop on this function call, invoke the `user_call()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `Bdb.quitting` flag is set (which can be set from `user_call()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

**dispatch_return**(frame, arg)
If the debugger should stop on this function return, invoke the `user_return()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `Bdb.quitting` flag is set (which can be set from `user_return()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

**dispatch_exception**(frame, arg)
If the debugger should stop at this exception, invokes the `user_exception()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `Bdb.quitting` flag is set (which can be set from `user_exception()`). Return a
reference to the `trace_dispatch()` method for further tracing in that scope.

Normally derived classes don’t override the following methods, but they may if they want to redefine the definition of stopping and breakpoints.

**stop_here(frame)**
This method checks if the `frame` is somewhere below `botframe` in the call stack. `botframe` is the frame in which debugging started.

**break_here(frame)**
This method checks if there is a breakpoint in the filename and line belonging to `frame` or, at least, in the current function. If the breakpoint is a temporary one, this method deletes it.

**break_anywhere(frame)**
This method checks if there is a breakpoint in the filename of the current frame.

Derived classes should override these methods to gain control over debugger operation.

**user_call(frame, argument_list)**
This method is called from `dispatch_call()` when there is the possibility that a break might be necessary anywhere inside the called function.

**user_line(frame)**
This method is called from `dispatch_line()` when either `stop_here()` or `break_here()` yields `True`.

**user_return(frame, return_value)**
This method is called from `dispatch_return()` when
stop_here() yields True.

user_exception(frame, exc_info)
    This method is called from dispatch_exception() when stop_here() yields True.

do_clear(arg)
    Handle how a breakpoint must be removed when it is a temporary one.

    This method must be implemented by derived classes.

Derived classes and clients can call the following methods to affect the stepping state.

set_step()
    Stop after one line of code.

set_next(frame)
    Stop on the next line in or below the given frame.

set_return(frame)
    Stop when returning from the given frame.

set_until(frame)
    Stop when the line with the line no greater than the current one is reached or when returning from current frame.

set_trace([frame])
    Start debugging from frame. If frame is not specified, debugging starts from caller’s frame.

set_continue()
    Stop only at breakpoints or when finished. If there are no breakpoints, set the system trace function to None.
set_quit()
    Set the quitting attribute to True. This raises BdbQuit in the next call to one of the dispatch_*( ) methods.

Derived classes and clients can call the following methods to manipulate breakpoints. These methods return a string containing an error message if something went wrong, or None if all is well.

set_break(filename, lineno, temporary=0, cond=None, funcname=None)
    Set a new breakpoint. If the lineno line doesn’t exist for the filename passed as argument, return an error message. The filename should be in canonical form, as described in the canonic() method.

clear_break(filename, lineno)
    Delete the breakpoints in filename and lineno. If none were set, an error message is returned.

clear_bpbynumber(arg)
    Delete the breakpoint which has the index arg in the Breakpoint.bpbynumber. If arg is not numeric or out of range, return an error message.

clear_all_file_breaks(filename)
    Delete all breakpoints in filename. If none were set, an error message is returned.

clear_all_breaks()
    Delete all existing breakpoints.

get_break(filename, lineno)
    Check if there is a breakpoint for lineno of filename.
get_breaks(filename, lineno)
Return all breakpoints for lineno in filename, or an empty list if none are set.

get_file_breaks(filename)
Return all breakpoints in filename, or an empty list if none are set.

get_all_breaks()
Return all breakpoints that are set.

Derived classes and clients can call the following methods to get a data structure representing a stack trace.

get_stack(f, t)
Get a list of records for a frame and all higher (calling) and lower frames, and the size of the higher part.

format_stack_entry(frame_lineno[], lprefix=': ')
Return a string with information about a stack entry, identified by a (frame, lineno) tuple:

- The canonical form of the filename which contains the frame.
- The function name, or "<lambda>".
- The input arguments.
- The return value.
- The line of code (if it exists).

The following two methods can be called by clients to use a debugger to debug a statement, given as a string.

run(cmd[, globals[, locals]])
Debug a statement executed via the exec statement. globals
defaults to \_\_main\_.\_\_dict\_, \_\_locals\_ defaults to \_\_globals\_.

\texttt{runeval}([\texttt{expr}[, \_\_globals\_[[, \_\_locals\_]]]])

Debug an expression executed via the \texttt{eval()} function. \_\_globals\_ and \_\_locals\_ have the same meaning as in \texttt{run()}.

\texttt{runcntx}([\texttt{cmd}, \_\_globals\_, \_\_locals\_])

For backwards compatibility. Calls the \texttt{run()} method.

\texttt{runcall}([\texttt{func}, \_\_args\_, \_\_kwds\_])

Debug a single function call, and return its result.

Finally, the module defines the following functions:

\texttt{bdb.\_\_checkfuncname\_\_}([\texttt{b}, \texttt{frame}])

Check whether we should break here, depending on the way the breakpoint \texttt{b} was set.

If it was set via line number, it checks if \texttt{b.line} is the same as the one in the frame also passed as argument. If the breakpoint was set via function name, we have to check we are in the right frame (the right function) and if we are in its first executable line.

\texttt{bdb.\_\_effective\_\_}([\texttt{file}, \texttt{line}, \texttt{frame}])

Determine if there is an effective (active) breakpoint at this line of code. Return a tuple of the breakpoint and a boolean that indicates if it is ok to delete a temporary breakpoint. Return (\texttt{None, None}) if there is no matching breakpoint.

\texttt{bdb.\_\_set\_\_trace\_\_}()

Start debugging with a \texttt{Bdb} instance from caller’s frame.
26.2. **pdb — The Python Debugger**

**Source code**: Lib/pdb.py

The module **pdb** defines an interactive source code debugger for Python programs. It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame. It also supports post-mortem debugging and can be called under program control.

The debugger is extensible — it is actually defined as the class **Pdb**. This is currently undocumented but easily understood by reading the source. The extension interface uses the modules **bdb** and **cmd**.

The debugger’s prompt is (**Pdb**). Typical usage to run a program under control of the debugger is:

```python
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

**pdb.py** can also be invoked as a script to debug other scripts. For example:

```bash
python -m pdb myscript.py
```

When invoked as a script, pdb will automatically enter post-mortem
debugging if the program being debugged exits abnormally. After post-mortem debugging (or after normal exit of the program), pdb will restart the program. Automatic restarting preserves pdb’s state (such as breakpoints) and in most cases is more useful than quitting the debugger upon program's exit.

*New in version 2.4:* Restarting post-mortem behavior added.

The typical usage to break into the debugger from a running program is to insert

```python
import pdb; pdb.set_trace()
```

at the location you want to break into the debugger. You can then step through the code following this statement, and continue running without the debugger using the `c` command.

The typical usage to inspect a crashed program is:

```python
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test
    print spam
NameError: name 'spam'
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print spam
(Pdb)
```

The module defines the following functions; each enters the debugger in a slightly different way:

```python
pdb.run(statement[, globals[, locals]])
```

Execute the `statement` (given as a string) under debugger
The debugger prompt appears before any code is executed; you can set breakpoints and type `continue`, or you can step through the statement using `step` or `next` (all these commands are explained below). The optional `globals` and `locals` arguments specify the environment in which the code is executed; by default the dictionary of the module `__main__` is used. (See the explanation of the `exec` statement or the `eval()` built-in function.)

```
pdb. runeval(expression[, globals[, locals]])
```
Evaluate the `expression` (given as a string) under debugger control. When `runeval()` returns, it returns the value of the expression. Otherwise this function is similar to `run()`.

```
pdb. runcall(function[, argument, ...])
```
Call the `function` (a function or method object, not a string) with the given arguments. When `runcall()` returns, it returns whatever the function call returned. The debugger prompt appears as soon as the function is entered.

```
pdb. set_trace()
```
Enter the debugger at the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails).

```
pdb. post_mortem([traceback])
```
Enter post-mortem debugging of the given `traceback` object. If no `traceback` is given, it uses the one of the exception that is currently being handled (an exception must be being handled if the default is to be used).

```
pdb. pm()
```
Enter post-mortem debugging of the traceback found in `sys.last_traceback`.

The `run*` functions and `set_trace()` are aliases for instantiating the `Pdb` class and calling the method of the same name. If you want to access further features, you have to do this yourself:

```python
class pdb.Pdb(completekey='tab', stdin=None, stdout=None, skip=None):
    Pdb is the debugger class.
```

The `completekey`, `stdin` and `stdout` arguments are passed to the underlying `cmd.Cmd` class; see the description there.

The `skip` argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. [1]

Example call to enable tracing with `skip`:

```python
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

*New in version 2.7:* The `skip` argument.

```python
run(statement[, globals[, locals]])
runeval(expression[, globals[, locals]])
runcall(function[, argument, ...])
set_trace()
```

See the documentation for the functions explained above.
The debugger recognizes the following commands. Most commands can be abbreviated to one or two letters; e.g. h(e1p) means that either h or help can be used to enter the help command (but not he or hel, nor H or Help or HELP). Arguments to commands must be separated by whitespace (spaces or tabs). Optional arguments are enclosed in square brackets ([ ]) in the command syntax; the square brackets must not be typed. Alternatives in the command syntax are separated by a vertical bar (|).

Entering a blank line repeats the last command entered. Exception: if the last command was a list command, the next 11 lines are listed.

Commands that the debugger doesn’t recognize are assumed to be Python statements and are executed in the context of the program being debugged. Python statements can also be prefixed with an exclamation point (!). This is a powerful way to inspect the program being debugged; it is even possible to change a variable or call a function. When an exception occurs in such a statement, the exception name is printed but the debugger’s state is not changed.

Multiple commands may be entered on a single line, separated by ;;. (A single ; is not used as it is the separator for multiple commands in a line that is passed to the Python parser.) No intelligence is applied to separating the commands; the input is split at the first ;; pair, even if it is in the middle of a quoted string.

The debugger supports aliases. Aliases can have parameters which allows one a certain level of adaptability to the context under examination.

If a file .pdbrc exists in the user’s home directory or in the current
directory, it is read in and executed as if it had been typed at the
debugger prompt. This is particularly useful for aliases. If both files
exist, the one in the home directory is read first and aliases defined
there can be overridden by the local file.

h(elp) [command]
Without argument, print the list of available commands. With a
command as argument, print help about that command. help pdb
displays the full documentation file; if the environment variable
PAGER is defined, the file is piped through that command instead.
Since the command argument must be an identifier, help exec
must be entered to get help on the ! command.

w(here)
Print a stack trace, with the most recent frame at the bottom. An
arrow indicates the current frame, which determines the context
of most commands.

d(own)
Move the current frame one level down in the stack trace (to a
newer frame).

u(p)
Move the current frame one level up in the stack trace (to an
older frame).

b(reak) [[filename:]lineno | function[, condition]]
With a lineno argument, set a break there in the current file. With
a function argument, set a break at the first executable statement
within that function. The line number may be prefixed with a
filename and a colon, to specify a breakpoint in another file
(probably one that hasn’t been loaded yet). The file is searched
on sys.path. Note that each breakpoint is assigned a number to
which all the other breakpoint commands refer.

If a second argument is present, it is an expression which must
evaluate to true before the breakpoint is honored.

Without argument, list all breaks, including for each breakpoint, the number of times that breakpoint has been hit, the current ignore count, and the associated condition if any.

tbreak [[filename:]lineno | function[, condition]]
    Temporary breakpoint, which is removed automatically when it is first hit. The arguments are the same as break.

cl(ear) [[filename:lineno | bpnumber [bpnumber ...]]
    With a filename:lineno argument, clear all the breakpoints at this line. With a space separated list of breakpoint numbers, clear those breakpoints. Without argument, clear all breaks (but first ask confirmation).

disable [bpnumber [bpnumber ...]]
    Disables the breakpoints given as a space separated list of breakpoint numbers. Disabling a breakpoint means it cannot cause the program to stop execution, but unlike clearing a breakpoint, it remains in the list of breakpoints and can be (re-)enabled.

enable [bpnumber [bpnumber ...]]
    Enables the breakpoints specified.

ignore bpnumber [count]
    Sets the ignore count for the given breakpoint number. If count is omitted, the ignore count is set to 0. A breakpoint becomes active when the ignore count is zero. When non-zero, the count is decremented each time the breakpoint is reached and the breakpoint is not disabled and any associated condition evaluates to true.

condition bpnumber [condition]
    Condition is an expression which must evaluate to true before the breakpoint is honored. If condition is absent, any existing condition is removed; i.e., the breakpoint is made unconditional.
commands [bpnumber]

Specify a list of commands for breakpoint number bpnumber. The commands themselves appear on the following lines. Type a line containing just ‘end’ to terminate the commands. An example:

```
(Pdb) commands 1
(com) print some_variable
(com) end
(Pdb)
```

To remove all commands from a breakpoint, type commands and follow it immediately with end; that is, give no commands.

With no bpnumber argument, commands refers to the last breakpoint set.

You can use breakpoint commands to start your program up again. Simply use the continue command, or step, or any other command that resumes execution.

Specifying any command resuming execution (currently continue, step, next, return, jump, quit and their abbreviations) terminates the command list (as if that command was immediately followed by end). This is because any time you resume execution (even with a simple next or step), you may encounter another breakpoint—which could have its own command list, leading to ambiguities about which list to execute.

If you use the ‘silent’ command in the command list, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If none of the other commands print anything, you see no sign that the breakpoint was reached.

*New in version 2.5.*

s(tep)
Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).

n(ext)
Continue execution until the next line in the current function is reached or it returns. (The difference between next and step is that step stops inside a called function, while next executes called functions at (nearly) full speed, only stopping at the next line in the current function.)

unt(il)
Continue execution until the line with the line number greater than the current one is reached or when returning from current frame.

*New in version 2.6.*

r(eturn)
Continue execution until the current function returns.

c(ont(inue))
Continue execution, only stop when a breakpoint is encountered.

j(ump) *lineno*
Set the next line that will be executed. Only available in the bottom-most frame. This lets you jump back and execute code again, or jump forward to skip code that you don't want to run.

It should be noted that not all jumps are allowed — for instance it is not possible to jump into the middle of a *for* loop or out of a *finally* clause.

l(ist) [*first*, *last*]
List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing. With one argument, list 11 lines around at that line. With two
arguments, list the given range; if the second argument is less than the first, it is interpreted as a count.

a(rgs)
Print the argument list of the current function.

p expression
Evaluate the expression in the current context and print its value.

**Note:** `print` can also be used, but is not a debugger command — this executes the Python `print` statement.

pp expression
Like the `p` command, except the value of the expression is pretty-printed using the `pprint` module.

alias [name [command]]
Creates an alias called `name` that executes `command`. The command must not be enclosed in quotes. Replaceable parameters can be indicated by `%1`, `%2`, and so on, while `%*` is replaced by all the parameters. If no command is given, the current alias for `name` is shown. If no arguments are given, all aliases are listed.

Aliases may be nested and can contain anything that can be legally typed at the pdb prompt. Note that internal pdb commands can be overridden by aliases. Such a command is then hidden until the alias is removed. Aliasing is recursively applied to the first word of the command line; all other words in the line are left alone.

As an example, here are two useful aliases (especially when placed in the `.pdbrc` file):

```python
#Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print "%1.%s","=%",%1.%s
#Print instance variables in self
```
alias ps pi self

unalias name

Deletes the specified alias.

[!]statement

Execute the (one-line) statement in the context of the current stack frame. The exclamation point can be omitted unless the first word of the statement resembles a debugger command. To set a global variable, you can prefix the assignment command with a global command on the same line, e.g.:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

run [args ...]

Restart the debugged Python program. If an argument is supplied, it is split with “shlex” and the result is used as the new sys.argv. History, breakpoints, actions and debugger options are preserved. “restart” is an alias for “run”.

New in version 2.6.

q(uит)

Quit from the debugger. The program being executed is aborted.

Footnotes

[1] Whether a frame is considered to originate in a certain module is determined by the __name__ in the frame globals.
26.4. The Python Profilers

**Source code:** `Lib/profile.py` and `Lib/pstats.py`
26.4.1. Introduction to the profilers

cProfile and profile provide deterministic profiling of Python programs. A profile is a set of statistics that describes how often and for how long various parts of the program executed. These statistics can be formatted into reports via the pstats module.

The Python standard library provides three different implementations of the same profiling interface:

1. cProfile is recommended for most users; it’s a C extension with reasonable overhead that makes it suitable for profiling long-running programs. Based on lsprof, contributed by Brett Rosen and Ted Czotter.

   New in version 2.5.

2. profile, a pure Python module whose interface is imitated by cProfile, but which adds significant overhead to profiled programs. If you’re trying to extend the profiler in some way, the task might be easier with this module. Originally designed and written by Jim Roskind.

   Changed in version 2.4: Now also reports the time spent in calls to built-in functions and methods.

3. hotshot was an experimental C module that focused on minimizing the overhead of profiling, at the expense of longer data post-processing times. It is no longer maintained and may be dropped in a future version of Python.

   Changed in version 2.5: The results should be more meaningful than in the past: the timing core contained a critical bug.
The `profile` and `cProfile` modules export the same interface, so they are mostly interchangeable; `cProfile` has a much lower overhead but is newer and might not be available on all systems. `cProfile` is really a compatibility layer on top of the internal `_lsprof` module. The `hotshot` module is reserved for specialized usage.

**Note:** The profiler modules are designed to provide an execution profile for a given program, not for benchmarking purposes (for that, there is `timeit` for reasonably accurate results). This particularly applies to benchmarking Python code against C code: the profilers introduce overhead for Python code, but not for C-level functions, and so the C code would seem faster than any Python one.
This section is provided for users that “don’t want to read the manual.” It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application.

To profile a function that takes a single argument, you can do:

```python
import cProfile
import re

cProfile.run('re.compile("foo|bar")')
```

(Use `profile` instead of `cProfile` if the latter is not available on your system.)

The above action would run `re.compile()` and print profile results like the following:

```
197 function calls (192 primitive calls) in 0.002 seconds
Ordered by: standard name
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
1    0.000    0.000    0.001    0.001  <string>:1(<module>)
1    0.000    0.000    0.001    0.001  re.py:212(compile)
1    0.000    0.000    0.001    0.001  re.py:268(_compile)
1    0.000    0.000    0.000    0.000  sre_compile.py:172(_
1    0.000    0.000    0.000    0.000  sre_compile.py:201(_
4    0.000    0.000    0.000    0.000  sre_compile.py:25(_i
3/1  0.000    0.000    0.000    0.000  sre_compile.py:33(_c
```

The first line indicates that 197 calls were monitored. Of those calls, 192 were `primitive`, meaning that the call was not induced via recursion. The next line: `Ordered by: standard name`, indicates that the text string in the far right column was used to sort the output. The column headings include:
ncalls
    for the number of calls,

tottime
    for the total time spent in the given function (and excluding time made in calls to sub-functions)

percall
    is the quotient of tottime divided by ncalls

cumtime
    is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recursive functions.

percall
    is the quotient of cumtime divided by primitive calls

filename:lineno(function)
    provides the respective data of each function

When there are two numbers in the first column (for example 3/1), it means that the function recursed. The second value is the number of primitive calls and the former is the total number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

Instead of printing the output at the end of the profile run, you can save the results to a file by specifying a filename to the run() function:

```python
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

The `pstats.Stats` class reads profile results from a file and formats them in various ways.
The file `cProfile` can also be invoked as a script to profile another script. For example:

```
python -m cProfile [-o output_file] [-s sort_order] myscript.py
```

- `-o` writes the profile results to a file instead of to stdout.
- `-s` specifies one of the `sort_stats()` sort values to sort the output by. This only applies when `-o` is not supplied.

The `pstats` module’s `Stats` class has a variety of methods for manipulating and printing the data saved into a profile results file:

```python
import pstats
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

The `strip_dirs()` method removed the extraneous path from all the module names. The `sort_stats()` method sorted all the entries according to the standard module/line/name string that is printed. The `print_stats()` method printed out all the statistics. You might try the following sort calls:

```python
p.sort_stats('name')
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```python
p.sort_stats('cumulative').print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.
If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```python
p.sort_stats('time').print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

```python
p.sort_stats('file').print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with `__init__` in them). As one final example, you could try:

```python
p.sort_stats('time', 'cum').print_stats(.5, 'init')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: `.5`) of its original size, then only lines containing `init` are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (p is still sorted according to the last criteria) do:

```python
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you’re going to have to read the manual, or guess what the following functions do:

```python
p.print_callees()
p.add('restats')
```
Invoked as a script, the `pstats` module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using `cmd`) and interactive help.
Both the `profile` and `cProfile` modules provide the following functions:

**profile.run**(command, filename=None, sort=-1)

This function takes a single argument that can be passed to the `exec()` function, and an optional file name. In all cases this routine executes:

```python
exec(command, __main__.__dict__, __main__.__dict__)
```

and gathers profiling statistics from the execution. If no file name is present, then this function automatically creates a `Stats` instance and prints a simple profiling report. If the sort value is specified it is passed to this `Stats` instance to control how the results are sorted.

**profile.runctx**(command, globals, locals, filename=None)

This function is similar to `run()`, with added arguments to supply the globals and locals dictionaries for the `command` string. This routine executes:

```python
exec(command, globals, locals)
```

and gathers profiling statistics as in the `run()` function above.

**class profile.Profile**(timer=None, timeunit=0.0, subcalls=True, builtins=True)

This class is normally only used if more precise control over profiling is needed than what the `cProfile.run()` function provides.
A custom timer can be supplied for measuring how long code takes to run via the `timer` argument. This must be a function that returns a single number representing the current time. If the number is an integer, the `timeunit` specifies a multiplier that specifies the duration of each unit of time. For example, if the timer returns times measured in thousands of seconds, the time unit would be `0.001`.

Directly using the `Profile` class allows formatting profile results without writing the profile data to a file:

```python
import cProfile, pstats, StringIO
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
s = StringIO.StringIO()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print s.getvalue()
```

- **enable()**
  
  Start collecting profiling data.

- **disable()**
  
  Stop collecting profiling data.

- **create_stats()**
  
  Stop collecting profiling data and record the results internally as the current profile.

- **print_stats(sort=-1)**
  
  Create a `Stats` object based on the current profile and print the results to stdout.

- **dump_stats(filename)**
Write the results of the current profile to *filename*.

**run**(cmd)
Profile the cmd via `exec()`.

**runc**tx(cmd, **globals, locals**)
Profile the cmd via `exec()` with the specified global and local environment.

**runcall**(func, *args, **kwargs)
Profile `func(*args, **kwargs)`
26.4.4. The stats Class

Analysis of the profiler data is done using the Stats class.

```python
class pstats.Stats(*filenames or profile, stream=sys.stdout)
```

This class constructor creates an instance of a “statistics object” from a filename (or list of filenames) or from a Profile instance. Output will be printed to the stream specified by stream.

The file selected by the above constructor must have been created by the corresponding version of profile or cProfile. To be specific, there is no file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers. If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing Stats object, the add() method can be used.

Instead of reading the profile data from a file, a cProfile.Profile or profile.Profile object can be used as the profile data source.

Stats objects have the following methods:

```python
strip_dirs()
```

This method for the Stats class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a “random” order, as it was just after object initialization and loading. If strip_dirs() causes two function names to be indistinguishable (they are
on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

**add(** *filenames*)

This method of the **Stats** class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of **profile.run()** or **cProfile.run()**. Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

**dump_stats**(filename)

Save the data loaded into the **Stats** object to a file named **filename**. The file is created if it does not exist, and is overwritten if it already exists. This is equivalent to the method of the same name on the **profile.Profile** and **cProfile.Profile** classes.

*New in version 2.3.*

**sort_stats**(**keys**)

This method modifies the **Stats** object by sorting it according to the supplied criteria. The argument is typically a string identifying the basis of a sort (example: 'time' or 'name').

When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, **sort_stats('name', 'file')** will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

Abbreviations can be used for any key names, as long as the
abbreviation is unambiguous. The following are the keys currently defined:

<table>
<thead>
<tr>
<th>Valid Arg</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'calls'</td>
<td>call count</td>
</tr>
<tr>
<td>'cumulative'</td>
<td>cumulative time</td>
</tr>
<tr>
<td>'cumtime'</td>
<td>cumulative time</td>
</tr>
<tr>
<td>'file'</td>
<td>file name</td>
</tr>
<tr>
<td>'filename'</td>
<td>file name</td>
</tr>
<tr>
<td>'module'</td>
<td>file name</td>
</tr>
<tr>
<td>'ncalls'</td>
<td>call count</td>
</tr>
<tr>
<td>'pcalls'</td>
<td>primitive call count</td>
</tr>
<tr>
<td>'line'</td>
<td>line number</td>
</tr>
<tr>
<td>'name'</td>
<td>function name</td>
</tr>
<tr>
<td>'nfl'</td>
<td>name/file/line</td>
</tr>
<tr>
<td>'stdname'</td>
<td>standard name</td>
</tr>
<tr>
<td>'time'</td>
<td>internal time</td>
</tr>
<tr>
<td>'tottime'</td>
<td>internal time</td>
</tr>
</tbody>
</table>

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (alphabetical). The subtle distinction between 'nfl' and 'stdname' is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, 'nfl' does a numeric compare of the line numbers. In fact, `sort_stats('nfl')` is the same as `sort_stats('name', 'file', 'line')`.

For backward-compatibility reasons, the numeric arguments `-1, 0, 1,` and `2` are permitted. They are interpreted as...
'stdname', 'calls', 'time', and 'cumulative' respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

**reverse_order()**

This method for the `Stats` class reverses the ordering of the basic list within the object. Note that by default ascending vs descending order is properly selected based on the sort key of choice.

**print_stats(*restrictions)**

This method for the `Stats` class prints out a report as described in the `profile.run()` definition.

The order of the printing is based on the last `sort_stats()` operation done on the object (subject to caveats in `add()` and `strip_dirs()`).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a regular expression (to pattern match the standard name that is printed. If several restrictions are provided, then they are applied sequentially. For example:

```
print_stats(.1, 'foo:')
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename `.*foo:`. In contrast, the command:

```
print_stats('foo:', .1)
```
would limit the list to all functions having file names `.*foo:`, and then proceed to only print the first 10% of them.

**print_callers(*restrictions)**

This method for the Stats class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by `print_stats()`, and the definition of the restricting argument is also identical. Each caller is reported on its own line. The format differs slightly depending on the profiler that produced the stats:

- With `profile`, a number is shown in parentheses after each caller to show how many times this specific call was made. For convenience, a second non-parenthesized number repeats the cumulative time spent in the function at the right.
- With `cProfile`, each caller is preceded by three numbers: the number of times this specific call was made, and the total and cumulative times spent in the current function while it was invoked by this specific caller.

**print_callees(*restrictions)**

This method for the Stats class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the `print_callers()` method.
26.4.5. What Is Deterministic Profiling?

Deterministic profiling is meant to reflect the fact that all function call, function return, and exception events are monitored, and precise timings are made for the intervals between these events (during which time the user’s code is executing). In contrast, statistical profiling (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required to do deterministic profiling. Python automatically provides a hook (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify “hot loops” that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.
26.4.6. Limitations

One limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying “clock” is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate than the underlying clock. If enough measurements are taken, then the “error” will tend to average out. Unfortunately, removing this first error induces a second source of error.

The second problem is that it “takes a while” from when an event is dispatched until the profiler’s call to get the time actually gets the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock’s value was obtained (and then squirreled away), until the user’s code is once again executing. As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (less than one clock tick), but it can accumulate and become very significant.

The problem is more important with profile than with the lower-overhead cProfile. For this reason, profile provides a means of calibrating itself for a given platform so that this error can be probabilistically (on the average) removed. After the profiler is calibrated, it will be more accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-) ). Do not be alarmed by negative numbers in the profile. They should only appear if you have calibrated your profiler, and the results are actually better than without calibration.
26.4.7. Calibration

The profiler of the `profile` module subtracts a constant from each event handling time to compensate for the overhead of calling the time function, and socking away the results. By default, the constant is 0. The following procedure can be used to obtain a better constant for a given platform (see Limitations).

```python
import profile
pr = profile.Profile()
for i in range(5):
    print pr.calibrate(10000)
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on a 1.8Ghz Intel Core i5 running Mac OS X, and using Python’s `time.clock()` as the timer, the magical number is about 4.04e-6.

The object of this exercise is to get a fairly consistent result. If your computer is very fast, or your timer function has poor resolution, you might have to pass 100000, or even 1000000, to get consistent results.

When you have a consistent answer, there are three ways you can use it: [1]

```python
import profile

# 1. Apply computed bias to all Profile instances created here
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias
```
# 3. Specify computed bias in instance constructor.

```python
pr = profile.Profile(bias=your_computed_bias)
```

If you have a choice, you are better off choosing a smaller constant, and then your results will “less often” show up as negative in profile statistics.
26.4.8. Using a custom timer

If you want to change how current time is determined (for example, to force use of wall-clock time or elapsed process time), pass the timing function you want to the Profile class constructor:

```
pr = profile.Profile(your_time_func)
```

The resulting profiler will then call `your_time_func`. Depending on whether you are using `profile.Profile` or `cProfile.Profile`, `your_time_func`'s return value will be interpreted differently:

**profile.Profile**

`your_time_func` should return a single number, or a list of numbers whose sum is the current time (like what `os.times()` returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the profiler class for the timer function that you choose (see Calibration). For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (`os.times()` is pretty bad, as it returns a tuple of floating point values). If you want to substitute a better timer in the cleanest fashion, derive a class and hardwire a replacement dispatch method that best handles your timer call, along with the appropriate calibration constant.

**cProfile.Profile**

`your_time_func` should return a single number. If it returns integers, you can also invoke the class constructor with a second argument specifying the real duration of one unit of time. For
example, if `your_integer_time_func` returns times measured in thousands of seconds, you would construct the `Profile` instance as follows:

```
pr = cProfile.Profile(your_integer_time_func, 0.001)
```

As the `cProfile.Profile` class cannot be calibrated, custom timer functions should be used with care and should be as fast as possible. For the best results with a custom timer, it might be necessary to hard-code it in the C source of the internal `_lsprof` module.

**Footnotes**

[1] Prior to Python 2.2, it was necessary to edit the profiler source code to embed the bias as a literal number. You still can, but that method is no longer described, because no longer needed.
26.5. hotshot — High performance logging profiler

New in version 2.2.

This module provides a nicer interface to the _hotshot C module. Hotshot is a replacement for the existing profile module. As it’s written mostly in C, it should result in a much smaller performance impact than the existing profile module.

Note: The hotshot module focuses on minimizing the overhead while profiling, at the expense of long data post-processing times. For common usage it is recommended to use cProfile instead. hotshot is not maintained and might be removed from the standard library in the future.

Changed in version 2.5: The results should be more meaningful than in the past: the timing core contained a critical bug.

Note: The hotshot profiler does not yet work well with threads. It is useful to use an unthreaded script to run the profiler over the code you’re interested in measuring if at all possible.

class hotshot. Profile(logfile[, lineevents[, linetimings]])

The profiler object. The argument logfile is the name of a log file to use for logged profile data. The argument lineevents specifies whether to generate events for every source line, or just on function call/return. It defaults to 0 (only log function call/return). The argument linetimings specifies whether to record timing information. It defaults to 1 (store timing information).
26.5.1. Profile Objects

Profile objects have the following methods:

Profile. `addinfo(key, value)`
Add an arbitrary labelled value to the profile output.

Profile. `close()`
Close the logfile and terminate the profiler.

Profile. `fileno()`
Return the file descriptor of the profiler’s log file.

Profile. `run(cmd)`
Profile an `exec`-compatible string in the script environment. The globals from the `__main__` module are used as both the globals and locals for the script.

Profile. `runcall(func, *args, **keywords)`
Profile a single call of a callable. Additional positional and keyword arguments may be passed along; the result of the call is returned, and exceptions are allowed to propagate cleanly, while ensuring that profiling is disabled on the way out.

Profile. `runctx(cmd, globals, locals)`
Evaluate an `exec`-compatible string in a specific environment. The string is compiled before profiling begins.

Profile. `start()`
Start the profiler.

Profile. `stop()`
Stop the profiler.
26.5.2. Using hotshot data

New in version 2.2.

This module loads hotshot profiling data into the standard pstats Stats objects.

hotshot.stats.load(filename)

Load hotshot data from filename. Returns an instance of the pstats.Stats class.

See also:

Module profile
The profile module's stats class
Note that this example runs the Python “benchmark” pystones. It can take some time to run, and will produce large output files.

```python
>>> import hotshot, hotshot.stats, test.pystone
>>> prof = hotshot.Profile("stones.prof")
>>> benchtime, stones = prof.runcall(test.pystone.pystones)
>>> prof.close()
>>> stats = hotshot.stats.load("stones.prof")
>>> stats.strip_dirs()
>>> stats.sort_stats('time', 'calls')
>>> stats.print_stats(20)
850004 function calls in 10.090 CPU seconds

Ordered by: internal time, call count

<table>
<thead>
<tr>
<th>ncalls</th>
<th>tottime</th>
<th>percall</th>
<th>cumtime</th>
<th>percall</th>
<th>filename:lineno(function)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.295</td>
<td>3.295</td>
<td>10.090</td>
<td>10.090</td>
<td>pystone.py:79(Proc0)</td>
</tr>
<tr>
<td>150000</td>
<td>1.315</td>
<td>0.000</td>
<td>1.315</td>
<td>0.000</td>
<td>pystone.py:203(Proc7)</td>
</tr>
<tr>
<td>50000</td>
<td>1.313</td>
<td>0.000</td>
<td>1.463</td>
<td>0.000</td>
<td>pystone.py:229(Func2)</td>
</tr>
</tbody>
</table>
```

26.6. timeit — Measure execution time of small code snippets

New in version 2.3.

**Source code:** Lib/timeit.py

This module provides a simple way to time small bits of Python code. It has both a Command-Line Interface as well as a callable one. It avoids a number of common traps for measuring execution times. See also Tim Peters' introduction to the “Algorithms” chapter in the *Python Cookbook*, published by O'Reilly.
26.6.1. Basic Examples

The following example shows how the Command-Line Interface can be used to compare three different expressions:

```
$ python -m timeit '"-.join(str(n) for n in range(100))'
10000 loops, best of 3: 40.3 usec per loop
$ python -m timeit '"-.join([str(n) for n in range(100)])'
10000 loops, best of 3: 33.4 usec per loop
$ python -m timeit '"-.join(map(str, range(100)))'
10000 loops, best of 3: 25.2 usec per loop
```

This can be achieved from the Python Interface with:

```
>>> import timeit
>>> timeit.timeit(""-.join(str(n) for n in range(100))", number=10000)
0.8187260627746582
>>> timeit.timeit(""-.join([str(n) for n in range(100)])", number=10000)
0.7288308143615723
>>> timeit.timeit(""-.join(map(str, range(100)))", number=10000)
0.5858950614929199
```

Note however that `timeit` will automatically determine the number of repetitions only when the command-line interface is used. In the Examples section you can find more advanced examples.
26.6.2. Python Interface

The module defines three convenience functions and a public class:

```python
timeit.timeit(stmt='pass', setup='pass', timer=<default timer>, number=1000000)
```

Create a `Timer` instance with the given statement, `setup` code and `timer` function and run its `timeit()` method with `number` executions.

*New in version 2.6.*

```python
timeit.repeat(stmt='pass', setup='pass', timer=<default timer>, repeat=3, number=1000000)
```

Create a `Timer` instance with the given statement, `setup` code and `timer` function and run its `repeat()` method with the given `repeat` count and `number` executions.

*New in version 2.6.*

```python
timeit.default_timer()
```

Define a default timer, in a platform-specific manner. On Windows, `time.clock()` has microsecond granularity, but `time.time()`'s granularity is 1/60th of a second. On Unix, `time.clock()` has 1/100th of a second granularity, and `time.time()` is much more precise. On either platform, `default_timer()` measures wall clock time, not the CPU time. This means that other processes running on the same computer may interfere with the timing.

```python
class timeit.Timer(stmt='pass', setup='pass', timer=<timer function>)
```

Class for timing execution speed of small code snippets.
The constructor takes a statement to be timed, an additional statement used for setup, and a timer function. Both statements default to 'pass'; the timer function is platform-dependent (see the module doc string). stmt and setup may also contain multiple statements separated by ; or newlines, as long as they don’t contain multi-line string literals.

To measure the execution time of the first statement, use the timeit() method. The repeat() method is a convenience to call timeit() multiple times and return a list of results.

Changed in version 2.6: The stmt and setup parameters can now also take objects that are callable without arguments. This will embed calls to them in a timer function that will then be executed by timeit(). Note that the timing overhead is a little larger in this case because of the extra function calls.

```
-timeit(number=1000000)
```

Time number executions of the main statement. This executes the setup statement once, and then returns the time it takes to execute the main statement a number of times, measured in seconds as a float. The argument is the number of times through the loop, defaulting to one million. The main statement, the setup statement and the timer function to be used are passed to the constructor.

Note: By default, timeit() temporarily turns off garbage collection during the timing. The advantage of this approach is that it makes independent timings more comparable. This disadvantage is that GC may be an important component of the performance of the function being measured. If so, GC can be re-enabled as the first statement in the setup string. For example:
repea**t**(repeat=3, number=1000000)
Call timeit() a few times.

This is a convenience function that calls the timeit() repeatedly, returning a list of results. The first argument specifies how many times to call timeit(). The second argument specifies the number argument for timeit().

**Note:** It’s tempting to calculate mean and standard deviation from the result vector and report these. However, this is not very useful. In a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python’s speed, but by other processes interfering with your timing accuracy. So the min() of the result is probably the only number you should be interested in. After that, you should look at the entire vector and apply common sense rather than statistics.

print_exc(file=None)
Helper to print a traceback from the timed code.

Typical use:

```python
t = Timer(...)  # outside the try/except
try:
    t.timeit(...)  # or t.repeat(...)
except:
    t.print_exc()
```

The advantage over the standard traceback is that source lines in the compiled template will be displayed. The optional
file argument directs where the traceback is sent; it defaults to 
sys.stderr.
26.6.3. Command-Line Interface

When called as a program from the command line, the following form is used:

```
python -m timeit [-n N] [-r N] [-s S] [-t] [-c] [-h] [statement]
```

Where the following options are understood:

- **-n N, --number=N**
  how many times to execute ‘statement’

- **-r N, --repeat=N**
  how many times to repeat the timer (default 3)

- **-s S, --setup=S**
  statement to be executed once initially (default pass)

- **-t, --time**
  use `time.time()` (default on all platforms but Windows)

- **-c, --clock**
  use `time.clock()` (default on Windows)

- **-v, --verbose**
  print raw timing results; repeat for more digits precision

- **-h, --help**
  print a short usage message and exit

A multi-line statement may be given by specifying each line as a separate statement argument; indented lines are possible by enclosing an argument in quotes and using leading spaces. Multiple `-s` options are treated similarly.
If \(-n\) is not given, a suitable number of loops is calculated by trying successive powers of 10 until the total time is at least 0.2 seconds.

`default_timer()` measurements can be affected by other programs running on the same machine, so the best thing to do when accurate timing is necessary is to repeat the timing a few times and use the best time. The `-r` option is good for this; the default of 3 repetitions is probably enough in most cases. On Unix, you can use `time.clock()` to measure CPU time.

**Note:** There is a certain baseline overhead associated with executing a pass statement. The code here doesn’t try to hide it, but you should be aware of it. The baseline overhead can be measured by invoking the program without arguments, and it might differ between Python versions. Also, to fairly compare older Python versions to Python 2.3, you may want to use Python’s `-O` option (see Optimizations) for the older versions to avoid timing SET_LINENO instructions.
26.6.4. Examples

It is possible to provide a setup statement that is executed only once at the beginning:

```python
import timeit

import timeit
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample string"'
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"'
1.7246671520006203

The same can be done using the Timer class and its methods:

```python
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"
0.395516149999312
>>> t.repeat()
[0.40193588800002544, 0.3960157959998014, 0.39594301399984033]

The following examples show how to time expressions that contain multiple lines. Here we compare the cost of using `hasattr()` vs. `try/except` to test for missing and present object attributes:

```python
$ python -m timeit 'try: ' str.__nonzero__' except AttributeError
100000 loops, best of 3: 15.7 usec per loop
$ python -m timeit 'if hasattr(str, "__nonzero__"): pass'
100000 loops, best of 3: 4.26 usec per loop

$ python -m timeit 'try: ' int.__nonzero__' except AttributeError
100000 loops, best of 3: 1.43 usec per loop
$ python -m timeit 'if hasattr(int, "__nonzero__"): pass'
```
To give the `timeit` module access to functions you define, you can pass a `setup` parameter which contains an import statement:

```python
def test():
    """Stupid test function""
    L = []
    for i in range(100):
        L.append(i)

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import t"
26.7. `trace` — Trace or track Python statement execution

**Source code:** `Lib/trace.py`

The `trace` module allows you to trace program execution, generate annotated statement coverage listings, print caller/callee relationships and list functions executed during a program run. It can be used in another program or from the command line.
26.7.1. Command-Line Usage

The `trace` module can be invoked from the command line. It can be as simple as

```
python -m trace --count -C . somefile.py ...
```

The above will execute `somefile.py` and generate annotated listings of all Python modules imported during the execution into the current directory.

**--help**
Display usage and exit.

**--version**
Display the version of the module and exit.

26.7.1.1. Main options

At least one of the following options must be specified when invoking `trace`. The `--listfuncs` option is mutually exclusive with the `--trace` and `--count` options. When `--listfuncs` is provided, neither `--count` nor `--trace` are accepted, and vice versa.

**-c, --count**
Produce a set of annotated listing files upon program completion that shows how many times each statement was executed. See also `--coverdir`, `--file` and `--no-report` below.

**-t, --trace**
Display lines as they are executed.

**-l, --listfuncs**
Display the functions executed by running the program.
-r, --report
  Produce an annotated list from an earlier program run that used the --count and --file option. This does not execute any code.

-T, --trackcalls
  Display the calling relationships exposed by running the program.

26.7.1.2. Modifiers

-f, --file=<file>
  Name of a file to accumulate counts over several tracing runs. Should be used with the --count option.

-C, --coverdir=<dir>
  Directory where the report files go. The coverage report for package.module is written to file dir/package/module.cover.

-m, --missing
  When generating annotated listings, mark lines which were not executed with >>>>>>.

-s, --summary
  When using --count or --report, write a brief summary to stdout for each file processed.

-R, --no-report
  Do not generate annotated listings. This is useful if you intend to make several runs with --count, and then produce a single set of annotated listings at the end.

-g, --timing
  Prefix each line with the time since the program started. Only used while tracing.

26.7.1.3. Filters
These options may be repeated multiple times.

**--ignore-module**\=<mod>

Ignore each of the given module names and its submodules (if it is a package). The argument can be a list of names separated by a comma.

**--ignore-dir**\=<dir>

Ignore all modules and packages in the named directory and subtdirectories. The argument can be a list of directories separated by *os.pathsep*. 
class trace. Trace([count=1[, trace=1[, countfuncs=0[, countcallers=0[, ignoremods=()[, ignoredirs=()[, infile=None[, outfile=None[, timing=False]]]]]]]]])

Create an object to trace execution of a single statement or expression. All parameters are optional. count enables counting of line numbers. trace enables line execution tracing. countfuncs enables listing of the functions called during the run. countcallers enables call relationship tracking. ignoremods is a list of modules or packages to ignore. ignoredirs is a list of directories whose modules or packages should be ignored. infile is the name of the file from which to read stored count information. outfile is the name of the file in which to write updated count information. timing enables a timestamp relative to when tracing information was started to be displayed.

run(cmd)

Execute the command and gather statistics from the execution with the current tracing parameters. cmd must be a string or code object, suitable for passing into exec().

runctx(cmd, globals=None, locals=None)

Execute the command and gather statistics from the execution with the current tracing parameters, in the defined global and local environments. If not defined, globals and locals default to empty dictionaries.

runfunc(func, *args, **kwds)

Call func with the given arguments under control of the Trace object with the current tracing parameters.
results()

Return a CoverageResults object that contains the cumulative results of all previous calls to run, runctx and runfunc for the given Trace instance. Does not reset the accumulated trace results.

class trace.CoverageResults

A container for coverage results, created by Trace.results(). Should not be created directly by the user.

update(other)

Merge in data from another CoverageResults object.

write_results([show_missing=True, summary=False, coverdir=None])

Write coverage results. Set show_missing to show lines that had no hits. Set summary to include in the output the coverage summary per module. coverdir specifies the directory into which the coverage result files will be output. If None, the results for each source file are placed in its directory.

A simple example demonstrating the use of the programmatic interface:

```
import sys
import trace

# create a Trace object, telling it what to ignore, and whether
# do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# run the new command using the given tracer
tracer.run('main()')
```
# make a report, placing output in the current directory
r = tracer.results()
```
r.write_results(show_missing=True, coverdir=".")
```
27. Software Packaging and Distribution

These libraries help you with publishing and installing Python software. While these modules are designed to work in conjunction with the Python Package Index, they can also be used with a local index server, or without any index server at all.

- **27.1. distutils** — Building and installing Python modules
- **27.2. ensurepip** — Bootstrapping the pip installer
  - 27.2.1. Command line interface
  - 27.2.2. Module API
27.1. distutils — Building and installing Python modules

The distutils package provides support for building and installing additional modules into a Python installation. The new modules may be either 100%-pure Python, or may be extension modules written in C, or may be collections of Python packages which include modules coded in both Python and C.

Most Python users will not want to use this module directly, but instead use the cross-version tools maintained by the Python Packaging Authority. In particular, setuptools is an enhanced alternative to distutils that provides:

- support for declaring project dependencies
- additional mechanisms for configuring which files to include in source releases (including plugins for integration with version control systems)
- the ability to declare project “entry points”, which can be used as the basis for application plugin systems
- the ability to automatically generate Windows command line executables at installation time rather than needing to prebuild them
- consistent behaviour across all supported Python versions

The recommended pip installer runs all setup.py scripts with setuptools, even if the script itself only imports distutils. Refer to the Python Packaging User Guide for more information.

For the benefits of packaging tool authors and users seeking a deeper understanding of the details of the current packaging and distribution system, the legacy distutils based user documentation and API reference remain available:
• Installing Python Modules (Legacy version)
• Distributing Python Modules (Legacy version)
27.2. ensurepip — Bootstrapping the pip installer

New in version 2.7.9.

The ensurepip package provides support for bootstrapping the pip installer into an existing Python installation or virtual environment. This bootstrapping approach reflects the fact that pip is an independent project with its own release cycle, and the latest available stable version is bundled with maintenance and feature releases of the CPython reference interpreter.

In most cases, end users of Python shouldn't need to invoke this module directly (as pip should be bootstrapped by default), but it may be needed if installing pip was skipped when installing Python (or when creating a virtual environment) or after explicitly uninstalling pip.

**Note:** This module does not access the internet. All of the components needed to bootstrap pip are included as internal parts of the package.

**See also:**

**Installing Python Modules**

The end user guide for installing Python packages

**PEP 453: Explicit bootstrapping of pip in Python installations**

The original rationale and specification for this module.

**PEP 477: Backport ensurepip (PEP 453) to Python 2.7**

The rationale and specification for backporting PEP 453 to Python 2.7.
27.2.1. Command line interface

The command line interface is invoked using the interpreter’s `--m` switch.

The simplest possible invocation is:

```
python -m ensurepip
```

This invocation will install `pip` if it is not already installed, but otherwise does nothing. To ensure the installed version of `pip` is at least as recent as the one bundled with `ensurepip`, pass the `--upgrade` option:

```
python -m ensurepip --upgrade
```

By default, `pip` is installed into the current virtual environment (if one is active) or into the system site packages (if there is no active virtual environment). The installation location can be controlled through two additional command line options:

- `--root <dir>`: Installs `pip` relative to the given root directory rather than the root of the currently active virtual environment (if any) or the default root for the current Python installation.
- `--user`: Installs `pip` into the user site packages directory rather than globally for the current Python installation (this option is not permitted inside an active virtual environment).

By default, the scripts `pip`, `pipX`, and `pipX.Y` will be installed (where `X.Y` stands for the version of Python used to invoke `ensurepip`). The scripts installed can be controlled through two additional command line options:
- `--altinstall`: if an alternate installation is requested, the `pip` and `pipx` script will *not* be installed.
- `--no-default-pip`: if a non-default installation is request, the `pip` script will *not* be installed.
27.2.2. Module API

`ensurepip` exposes two functions for programmatic use:

```python
ensurepip.version()
```

Returns a string specifying the bundled version of pip that will be installed when bootstrapping an environment.

```python
ensurepip.bootstrap(root=None, upgrade=False, user=False, altinstall=False, default_pip=True, verbosity=0)
```

Bootstraps pip into the current or designated environment.

- `root` specifies an alternative root directory to install relative to. If `root` is `None`, then installation uses the default install location for the current environment.

- `upgrade` indicates whether or not to upgrade an existing installation of an earlier version of pip to the bundled version.

- `user` indicates whether to use the user scheme rather than installing globally.

By default, the scripts `pip`, `pipX`, and `pipX.Y` will be installed (where X.Y stands for the current version of Python).

If `altinstall` is set, then `pip` and `pipX` will not be installed.

If `default_pip` is set to `False`, then `pip` will not be installed.

Setting both `altinstall` and `default_pip` will trigger `ValueError`.

`verbosity` controls the level of output to `sys.stdout` from the bootstrapping operation.
**Note:** The bootstrapping process has side effects on both `sys.path` and `os.environ`. Invoking the command line interface in a subprocess instead allows these side effects to be avoided.

**Note:** The bootstrapping process may install additional modules required by `pip`, but other software should not assume those dependencies will always be present by default (as the dependencies may be removed in a future version of `pip`).
28. Python Runtime Services

The modules described in this chapter provide a wide range of services related to the Python interpreter and its interaction with its environment. Here's an overview:

- **28.1. `sys`** — System-specific parameters and functions
- **28.2. `sysconfig`** — Provide access to Python's configuration information
  - 28.2.1. Configuration variables
  - 28.2.2. Installation paths
  - 28.2.3. Other functions
- **28.3. `__builtin__`** — Built-in objects
- **28.4. `future_builtins`** — Python 3 builtins
- **28.5. `__main__`** — Top-level script environment
- **28.6. `warnings`** — Warning control
  - 28.6.1. Warning Categories
  - 28.6.2. The Warnings Filter
    - 28.6.2.1. Default Warning Filters
  - 28.6.3. Temporarily Suppressing Warnings
  - 28.6.4. Testing Warnings
  - 28.6.5. Updating Code For New Versions of Python
  - 28.6.6. Available Functions
  - 28.6.7. Available Context Managers
- **28.7. `contextlib`** — Utilities for with-statement contexts
- **28.8. `abc`** — Abstract Base Classes
- **28.9. `atexit`** — Exit handlers
  - 28.9.1. `atexit` Example
- **28.10. `traceback`** — Print or retrieve a stack traceback
  - 28.10.1. Traceback Examples
- **28.11. `__future__`** — Future statement definitions
- **28.12. `gc`** — Garbage Collector interface
• 28.13. inspect — Inspect live objects
  o 28.13.1. Types and members
  o 28.13.2. Retrieving source code
  o 28.13.3. Classes and functions
  o 28.13.4. The interpreter stack
• 28.14. site — Site-specific configuration hook
• 28.15. user — User-specific configuration hook
• 28.16. fpectl — Floating point exception control
  o 28.16.1. Example
  o 28.16.2. Limitations and other considerations
This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.

**sys.argv**

The list of command line arguments passed to a Python script. `argv[0]` is the script name (it is operating system dependent whether this is a full pathname or not). If the command was executed using the `-c` command line option to the interpreter, `argv[0]` is set to the string `-c`. If no script name was passed to the Python interpreter, `argv[0]` is the empty string.

To loop over the standard input, or the list of files given on the command line, see the `fileinput` module.

**sys.byteorder**

An indicator of the native byte order. This will have the value 'big' on big-endian (most-significant byte first) platforms, and 'little' on little-endian (least-significant byte first) platforms.

*New in version 2.0.*

**sys.builtin_module_names**

A tuple of strings giving the names of all modules that are compiled into this Python interpreter. (This information is not available in any other way — `modules.keys()` only lists the imported modules.)

**sys.call_tracing(func, args)**

Call `func(*args)`, while tracing is enabled. The tracing state is
saved, and restored afterwards. This is intended to be called from a debugger from a checkpoint, to recursively debug some other code.

sys.\texttt{copyright}
A string containing the copyright pertaining to the Python interpreter.

sys.\texttt{\_clear\_type\_cache()}
Clear the internal type cache. The type cache is used to speed up attribute and method lookups. Use the function \texttt{only} to drop unnecessary references during reference leak debugging.

This function should be used for internal and specialized purposes only.

\textit{New in version 2.6.}

sys.\texttt{\_current\_frames()}
Return a dictionary mapping each thread's identifier to the topmost stack frame currently active in that thread at the time the function is called. Note that functions in the \texttt{traceback} module can build the call stack given such a frame.

This is most useful for debugging deadlock: this function does not require the deadlocked threads' cooperation, and such threads' call stacks are frozen for as long as they remain deadlocked. The frame returned for a non-deadlocked thread may bear no relationship to that thread's current activity by the time calling code examines the frame.

This function should be used for internal and specialized purposes only.

\textit{New in version 2.5.}
**sys.dllhandle**

Integer specifying the handle of the Python DLL. Availability: Windows.

**sys.displayhook(value)**

If `value` is not `None`, this function prints it to `sys.stdout`, and saves it in `__builtin__.__displayhook__`.

`sys.displayhook` is called on the result of evaluating an expression entered in an interactive Python session. The display of these values can be customized by assigning another one-argument function to `sys.displayhook`.

**sys.dont_write_bytecode**

If this is true, Python won’t try to write `.pyc` or `.pyo` files on the import of source modules. This value is initially set to `True` or `False` depending on the `-B` command line option and the `PYTHONDONTRCWRITEBYTECODE` environment variable, but you can set it yourself to control bytecode file generation.

*New in version 2.6.*

**sys.excepthook(type, value, traceback)**

This function prints out a given traceback and exception to `sys.stderr`.

When an exception is raised and uncaught, the interpreter calls `sys.excepthook` with three arguments, the exception class, exception instance, and a traceback object. In an interactive session this happens just before control is returned to the prompt; in a Python program this happens just before the program exits. The handling of such top-level exceptions can be customized by assigning another three-argument function to `sys.excepthook`.

**sys.__displayhook__**
**sys.__excepthook__**

These objects contain the original values of `displayhook` and `excepthook` at the start of the program. They are saved so that `displayhook` and `excepthook` can be restored in case they happen to get replaced with broken objects.

**sys.exc_info()**

This function returns a tuple of three values that give information about the exception that is currently being handled. The information returned is specific both to the current thread and to the current stack frame. If the current stack frame is not handling an exception, the information is taken from the calling stack frame, or its caller, and so on until a stack frame is found that is handling an exception. Here, “handling an exception” is defined as “executing or having executed an except clause.” For any stack frame, only information about the most recently handled exception is accessible.

If no exception is being handled anywhere on the stack, a tuple containing three `None` values is returned. Otherwise, the values returned are `(type, value, traceback)`. Their meaning is: *type* gets the exception type of the exception being handled (a class object); *value* gets the exception parameter (its associated value or the second argument to `raise`, which is always a class instance if the exception type is a class object); *traceback* gets a traceback object (see the Reference Manual) which encapsulates the call stack at the point where the exception originally occurred.

If `exc_clear()` is called, this function will return three `None` values until either another exception is raised in the current thread or the execution stack returns to a frame where another exception is being handled.

**Warning:** Assigning the *traceback* return value to a local
variable in a function that is handling an exception will cause a circular reference. This will prevent anything referenced by a local variable in the same function or by the traceback from being garbage collected. Since most functions don't need access to the traceback, the best solution is to use something like `exctype, value = sys.exc_info()[:2]` to extract only the exception type and value. If you do need the traceback, make sure to delete it after use (best done with a `try ... finally` statement) or to call `exc_info()` in a function that does not itself handle an exception.

**Note:** Beginning with Python 2.2, such cycles are automatically reclaimed when garbage collection is enabled and they become unreachable, but it remains more efficient to avoid creating cycles.

```python
sys.exc_clear()
```

This function clears all information relating to the current or last exception that occurred in the current thread. After calling this function, `exc_info()` will return three `None` values until another exception is raised in the current thread or the execution stack returns to a frame where another exception is being handled.

This function is only needed in only a few obscure situations. These include logging and error handling systems that report information on the last or current exception. This function can also be used to try to free resources and trigger object finalization, though no guarantee is made as to what objects will be freed, if any.

*New in version 2.3.*

```python
sys.exc_type
sys.exc_value
sys.exc_traceback
```
Deprecated since version 1.5: Use `exc_info()` instead.

Since they are global variables, they are not specific to the current thread, so their use is not safe in a multi-threaded program. When no exception is being handled, `exc_type` is set to `None` and the other two are undefined.

**sys.exec_prefix**

A string giving the site-specific directory prefix where the platform-dependent Python files are installed; by default, this is also `'/usr/local'`. This can be set at build time with the `--exec-prefix` argument to the `configure` script. Specifically, all configuration files (e.g. the `pyconfig.h` header file) are installed in the directory `exec_prefix/lib/pythonX.Y/config`, and shared library modules are installed in `exec_prefix/lib/pythonX.Y/lib-dynload`, where `X.Y` is the version number of Python, for example 2.7.

**sys.executable**

A string giving the absolute path of the executable binary for the Python interpreter, on systems where this makes sense. If Python is unable to retrieve the real path to its executable, `sys.executable` will be an empty string or `None`.

**sys.exit([arg])**

Exit from Python. This is implemented by raising the `SystemExit` exception, so cleanup actions specified by finally clauses of `try` statements are honored, and it is possible to intercept the exit attempt at an outer level.

The optional argument `arg` can be an integer giving the exit status (defaulting to zero), or another type of object. If it is an integer, zero is considered “successful termination” and any nonzero value is considered “abnormal termination” by shells and
the like. Most systems require it to be in the range 0–127, and produce undefined results otherwise. Some systems have a convention for assigning specific meanings to specific exit codes, but these are generally underdeveloped; Unix programs generally use 2 for command line syntax errors and 1 for all other kind of errors. If another type of object is passed, None is equivalent to passing zero, and any other object is printed to stderr and results in an exit code of 1. In particular, `sys.exit("some error message")` is a quick way to exit a program when an error occurs.

Since `exit()` ultimately “only" raises an exception, it will only exit the process when called from the main thread, and the exception is not intercepted.

**sys.exitfunc**

This value is not actually defined by the module, but can be set by the user (or by a program) to specify a clean-up action at program exit. When set, it should be a parameterless function. This function will be called when the interpreter exits. Only one function may be installed in this way; to allow multiple functions which will be called at termination, use the `atexit` module.

**Note:** The exit function is not called when the program is killed by a signal, when a Python fatal internal error is detected, or when `os._exit()` is called.

*Deprecated since version 2.4:* Use `atexit` instead.

**sys.flags**

The struct sequence `flags` exposes the status of command line flags. The attributes are read only.

<table>
<thead>
<tr>
<th>attribute</th>
<th>flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug</td>
<td>-d</td>
</tr>
</tbody>
</table>
New in version 2.6.

New in version 2.7.3: The hash_randomization attribute.

sys. `float_info`
A structseq holding information about the float type. It contains low level information about the precision and internal representation. The values correspond to the various floating-point constants defined in the standard header file `float.h` for the ‘C’ programming language; see section 5.2.4.2.2 of the 1999 ISO/IEC C standard [C99], ‘Characteristics of floating types’, for details.

<table>
<thead>
<tr>
<th>attribute</th>
<th>float.h macro</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>epsilon</td>
<td>DBL_EPSILON</td>
<td>difference between 1 and the least value greater than 1 that is representable as a float</td>
</tr>
<tr>
<td>dig</td>
<td>DBL_DIG</td>
<td>maximum number of decimal digits that can be faithfully represented in a float</td>
</tr>
<tr>
<td>Attribute</td>
<td>Macro</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mant_dig</td>
<td>DBL_MANT_DIG</td>
<td>float precision: the number of base-radix digits in the significand of a float</td>
</tr>
<tr>
<td>max</td>
<td>DBL_MAX</td>
<td>maximum representable finite float</td>
</tr>
<tr>
<td>max_exp</td>
<td>DBL_MAX_EXP</td>
<td>maximum integer e such that ( \text{radix}^{(e-1)} ) is a representable finite float</td>
</tr>
<tr>
<td>max_10_exp</td>
<td>DBL_MAX_10_EXP</td>
<td>maximum integer e such that ( 10^{e} ) is in the range of representable finite floats</td>
</tr>
<tr>
<td>min</td>
<td>DBL_MIN</td>
<td>minimum positive normalized float</td>
</tr>
<tr>
<td>min_exp</td>
<td>DBL_MIN_EXP</td>
<td>minimum integer e such that ( \text{radix}^{(e-1)} ) is a normalized float</td>
</tr>
<tr>
<td>min_10_exp</td>
<td>DBL_MIN_10_EXP</td>
<td>minimum integer e such that ( 10^{e} ) is a normalized float</td>
</tr>
<tr>
<td>radix</td>
<td>FLT_RADIX</td>
<td>radix of exponent representation</td>
</tr>
<tr>
<td>rounds</td>
<td>FLT_ROUNDS</td>
<td>integer constant representing the rounding mode used for arithmetic operations. This reflects the value of the system FLT_ROUNDS macro at interpreter startup time. See section 5.2.4.2.2 of the C99 standard for an explanation of the possible values and their meanings.</td>
</tr>
</tbody>
</table>

The attribute `sys.float_info.dig` needs further explanation. If \( s \) is any string representing a decimal number with at most `sys.float_info.dig` significant digits, then converting \( s \) to a float
and back again will recover a string representing the same decimal value:

```python
>>> import sys
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'  # decimal string with 15 significant digits
>>> format(float(s), '.15g')  # convert to float and back ->
'3.14159265358979'
```

But for strings with more than `sys.float_info.dig` significant digits, this isn’t always true:

```python
>>> s = '9876543211234567'  # 16 significant digits is too many!
>>> format(float(s), '.16g')  # conversion changes value
'9876543211234568'
```

*New in version 2.6.*

**sys.float_repr_style**

A string indicating how the `repr()` function behaves for floats. If the string has value `'short'` then for a finite float `x`, `repr(x)` aims to produce a short string with the property that `float(repr(x)) == x`. This is the usual behaviour in Python 2.7 and later. Otherwise, `float_repr_style` has value `'legacy'` and `repr(x)` behaves in the same way as it did in versions of Python prior to 2.7.

*New in version 2.7.*

**sys.getcheckinterval()**

Return the interpreter’s “check interval”; see `setcheckinterval()`.

*New in version 2.3.*

**sys.getdefaultencoding()**

Return the name of the current default string encoding used by
the Unicode implementation.

*New in version 2.0.*

c**sys. getdlopenflags()**
Return the current value of the flags that are used for `dlopen()` calls. The flag constants are defined in the `dl` and `DLFCN` modules. Availability: Unix.

*New in version 2.2.*

c**sys. getfilesystemencoding()**
Return the name of the encoding used to convert Unicode filenames into system file names, or `None` if the system default encoding is used. The result value depends on the operating system:

- On Mac OS X, the encoding is `'utf-8'`.
- On Unix, the encoding is the user’s preference according to the result of `nl_langinfo(CODESET)`, or `None` if the `nl_langinfo(CODESET)` failed.
- On Windows NT+, file names are Unicode natively, so no conversion is performed. `getfilesystemencoding()` still returns `'mbcs'`, as this is the encoding that applications should use when they explicitly want to convert Unicode strings to byte strings that are equivalent when used as file names.
- On Windows 9x, the encoding is `'mbcs'`.

*New in version 2.3.*

c**sys. getrefcount(object)**
Return the reference count of the object. The count returned is generally one higher than you might expect, because it includes the (temporary) reference as an argument to `getrefcount()`.
sys.\texttt{getrecursionlimit()}  
Return the current value of the recursion limit, the maximum depth of the Python interpreter stack. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python. It can be set by \texttt{setrecursionlimit()}.  

sys.\texttt{getsizeof}(\texttt{object[, default]})  
Return the size of an object in bytes. The object can be any type of object. All built-in objects will return correct results, but this does not have to hold true for third-party extensions as it is implementation specific.  

If given, \texttt{default} will be returned if the object does not provide means to retrieve the size. Otherwise a \texttt{TypeError} will be raised.  

\texttt{getsizeof()} calls the object's \texttt{\_\_sizeof\_} method and adds an additional garbage collector overhead if the object is managed by the garbage collector.  

\textit{New in version 2.6.}  

sys.\texttt{\_\_getframe\_(\texttt{depth})}  
Return a frame object from the call stack. If optional integer \texttt{depth} is given, return the frame object that many calls below the top of the stack. If that is deeper than the call stack, \texttt{ValueError} is raised. The default for \texttt{depth} is zero, returning the frame at the top of the call stack.  

\textbf{CPython implementation detail:} This function should be used for internal and specialized purposes only. It is not guaranteed to exist in all implementations of Python.  

sys.\texttt{getprofile()}
Get the profiler function as set by `setprofile()`.

*New in version 2.6.*

**sys.gettrace()**

Get the trace function as set by `settrace()`.

**CPython implementation detail:** The `gettrace()` function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

*New in version 2.6.*

**sys.getwindowsversion()**

Return a named tuple describing the Windows version currently running. The named elements are `major`, `minor`, `build`, `platform`, `service_pack`, `service_pack_minor`, `service_pack_major`, `suite_mask`, and `product_type`. `service_pack` contains a string while all other values are integers. The components can also be accessed by name, so `sys.getwindowsversion()[0]` is equivalent to `sys.getwindowsversion().major`. For compatibility with prior versions, only the first 5 elements are retrievable by indexing.

`platform` may be one of the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (VER_PLATFORM_WIN32s)</td>
<td>Win32s on Windows 3.1</td>
</tr>
<tr>
<td>1 (VER_PLATFORM_WIN32_WINDOWS)</td>
<td>Windows 95/98/ME</td>
</tr>
<tr>
<td>2 (VER_PLATFORM_WIN32_NT)</td>
<td>Windows NT/2000/XP/x64</td>
</tr>
<tr>
<td>3 (VER_PLATFORM_WIN32_CE)</td>
<td>Windows CE</td>
</tr>
</tbody>
</table>
*product_type* may be one of the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (VER_NT_WORKSTATION)</td>
<td>The system is a workstation.</td>
</tr>
<tr>
<td>2 (VER_NT_DOMAIN_CONTROLLER)</td>
<td>The system is a domain controller.</td>
</tr>
<tr>
<td>3 (VER_NT_SERVER)</td>
<td>The system is a server, but not a domain controller.</td>
</tr>
</tbody>
</table>

This function wraps the Win32 `GetVersionEx()` function; see the Microsoft documentation on `OSVERSIONINFOEX()` for more information about these fields.

Availability: Windows.

*New in version 2.3.*

*Changed in version 2.7:* Changed to a named tuple and added `service_pack_minor`, `service_pack_major`, `suite_mask`, and `product_type`.

**sys.hexversion**

The version number encoded as a single integer. This is guaranteed to increase with each version, including proper support for non-production releases. For example, to test that the Python interpreter is at least version 1.5.2, use:

```python
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

This is called `hexversion` since it only really looks meaningful when viewed as the result of passing it to the built-in `hex()` function. The `version_info` value may be used for a more
human-friendly encoding of the same information.

The `hexversion` is a 32-bit number with the following layout:

<table>
<thead>
<tr>
<th>Bits (big endian order)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-8</td>
<td><code>PY_MAJOR_VERSION</code> (the 2 in 2.1.0a3)</td>
</tr>
<tr>
<td>9-16</td>
<td><code>PY_MINOR_VERSION</code> (the 1 in 2.1.0a3)</td>
</tr>
<tr>
<td>17-24</td>
<td><code>PY_MICRO_VERSION</code> (the 0 in 2.1.0a3)</td>
</tr>
<tr>
<td>25-28</td>
<td><code>PY_RELEASE_LEVEL</code> (0xA for alpha, 0xB for beta, 0xC for release candidate and 0xF for final)</td>
</tr>
<tr>
<td>29-32</td>
<td><code>PY_RELEASE_SERIAL</code> (the 3 in 2.1.0a3, zero for final releases)</td>
</tr>
</tbody>
</table>

Thus 2.1.0a3 is hexversion 0x020100a3.

**New in version 1.5.2.**

`sys.long_info`

A struct sequence that holds information about Python's internal representation of integers. The attributes are read only.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bits_per_digit</code></td>
<td>number of bits held in each digit. Python integers are stored internally in base $2^{\text{long_info.bits_per_digit}}$</td>
</tr>
<tr>
<td><code>sizeof_digit</code></td>
<td>size in bytes of the C type used to represent a digit</td>
</tr>
</tbody>
</table>

**New in version 2.7.**

`sys.last_type`
`sys.last_value`
`sys.last_traceback`

These three variables are not always defined; they are set when an exception is not handled and the interpreter prints an error
message and a stack traceback. Their intended use is to allow an interactive user to import a debugger module and engage in post-mortem debugging without having to re-execute the command that caused the error. (Typical use is import pdb; pdb.pm() to enter the post-mortem debugger; see chapter pdb — The Python Debugger for more information.)

The meaning of the variables is the same as that of the return values from exc_info() above. (Since there is only one interactive thread, thread-safety is not a concern for these variables, unlike for exc_type etc.)

**sys.maxint**

The largest positive integer supported by Python’s regular integer type. This is at least 2**31-1. The largest negative integer is -\text{maxint-1} — the asymmetry results from the use of 2's complement binary arithmetic.

**sys.maxsize**

The largest positive integer supported by the platform’s Py_ssize_t type, and thus the maximum size lists, strings, dicts, and many other containers can have.

**sys.maxunicode**

An integer giving the largest supported code point for a Unicode character. The value of this depends on the configuration option that specifies whether Unicode characters are stored as UCS-2 or UCS-4.

**sys.meta_path**

A list of finder objects that have their find_module() methods called to see if one of the objects can find the module to be imported. The find_module() method is called at least with the absolute name of the module being imported. If the module to be imported is contained in package then the parent package's
__path__ attribute is passed in as a second argument. The method returns None if the module cannot be found, else returns a loader.

sys.meta_path is searched before any implicit default finders or sys.path.

See PEP 302 for the original specification.

sys.modules
This is a dictionary that maps module names to modules which have already been loaded. This can be manipulated to force reloading of modules and other tricks. Note that removing a module from this dictionary is not the same as calling reload() on the corresponding module object.

sys.path
A list of strings that specifies the search path for modules. Initialized from the environment variable PYTHONPATH, plus an installation-dependent default.

As initialized upon program startup, the first item of this list, path[0], is the directory containing the script that was used to invoke the Python interpreter. If the script directory is not available (e.g. if the interpreter is invoked interactively or if the script is read from standard input), path[0] is the empty string, which directs Python to search modules in the current directory first. Notice that the script directory is inserted before the entries inserted as a result of PYTHONPATH.

A program is free to modify this list for its own purposes.

Changed in version 2.3: Unicode strings are no longer ignored.

See also: Module site This describes how to use .pth files to
extend sys.path.

sys. **path_hooks**
A list of callables that take a path argument to try to create a finder for the path. If a finder can be created, it is to be returned by the callable, else raise ImportError.

Originally specified in **PEP 302**.

sys. **path_importer_cache**
A dictionary acting as a cache for finder objects. The keys are paths that have been passed to `sys.path_hooks` and the values are the finders that are found. If a path is a valid file system path but no explicit finder is found on `sys.path_hooks` then `None` is stored to represent the implicit default finder should be used. If the path is not an existing path then `imp.NullImporter` is set.

Originally specified in **PEP 302**.

sys. **platform**
This string contains a platform identifier that can be used to append platform-specific components to `sys.path`, for instance.

For most Unix systems, this is the lowercased OS name as returned by `uname -s` with the first part of the version as returned by `uname -r` appended, e.g. 'sunos5', *at the time when Python was built*. Unless you want to test for a specific system version, it is therefore recommended to use the following idiom:

```python
if sys.platform.startswith('freebsd'):
    # FreeBSD-specific code here...
elif sys.platform.startswith('linux'):
    # Linux-specific code here...
```

*Changed in version 2.7.3*: Since lots of code check for `sys.platform == 'linux2'`, and there is no essential change
between Linux 2.x and 3.x, `sys.platform` is always set to 'linux2', even on Linux 3.x. In Python 3.3 and later, the value will always be set to 'linux', so it is recommended to always use the `startswith` idiom presented above.

For other systems, the values are:

<table>
<thead>
<tr>
<th>System</th>
<th>Platform value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux (2.x and 3.x)</td>
<td>'linux2'</td>
</tr>
<tr>
<td>Windows</td>
<td>'win32'</td>
</tr>
<tr>
<td>Windows/Cygwin</td>
<td>'cygwin'</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>'darwin'</td>
</tr>
<tr>
<td>OS/2</td>
<td>'os2'</td>
</tr>
<tr>
<td>OS/2 EMX</td>
<td>'os2emx'</td>
</tr>
<tr>
<td>RiscOS</td>
<td>'riscos'</td>
</tr>
<tr>
<td>AtheOS</td>
<td>'atheos'</td>
</tr>
</tbody>
</table>

See also: `os.name` has a coarser granularity. `os.uname()` gives system-dependent version information. The `platform` module provides detailed checks for the system's identity.

**sys.prefix**

A string giving the site-specific directory prefix where the platform independent Python files are installed; by default, this is the string '/usr/local'. This can be set at build time with the `--prefix` argument to the `configure` script. The main collection of Python library modules is installed in the directory `prefix/lib/pythonX.Y` while the platform independent header files (all except `pyconfig.h`) are stored in `prefix/include/pythonX.Y`, where `X.Y` is the version number of Python, for example 2.7.

**sys.ps1**
**sys.ps2**

Strings specifying the primary and secondary prompt of the interpreter. These are only defined if the interpreter is in interactive mode. Their initial values in this case are '>>> ' and '. . . '. If a non-string object is assigned to either variable, its `str()` is re-evaluated each time the interpreter prepares to read a new interactive command; this can be used to implement a dynamic prompt.

**sys.py3kwarning**

Bool containing the status of the Python 3 warning flag. It’s `True` when Python is started with the `-3` option. (This should be considered read-only; setting it to a different value doesn't have an effect on Python 3 warnings.)

*New in version 2.6.*

**sys.setcheckinterval(interval)**

Set the interpreter’s “check interval”. This integer value determines how often the interpreter checks for periodic things such as thread switches and signal handlers. The default is `100`, meaning the check is performed every 100 Python virtual instructions. Setting it to a larger value may increase performance for programs using threads. Setting it to a value `<= 0` checks every virtual instruction, maximizing responsiveness as well as overhead.

**sys.setdefaultencoding(name)**

Set the current default string encoding used by the Unicode implementation. If `name` does not match any available encoding, `LookupError` is raised. This function is only intended to be used by the `site` module implementation and, where needed, by `sitecustomize`. Once used by the `site` module, it is removed from the `sys` module’s namespace.
New in version 2.0.

sys.setdlopenflags(n)
Set the flags used by the interpreter for dlopen() calls, such as when the interpreter loads extension modules. Among other things, this will enable a lazy resolving of symbols when importing a module, if called as sys.setdlopenflags(0). To share symbols across extension modules, call as sys.setdlopenflags(dl.RTLD_NOW | dl.RTLD_GLOBAL). Symbolic names for the flag modules can be either found in the dl module, or in the DLFCN module. If DLFCN is not available, it can be generated from /usr/include/dlfcn.h using the h2py script. Availability: Unix.

New in version 2.2.

sys.setprofile(profilefunc)
Set the system’s profile function, which allows you to implement a Python source code profiler in Python. See chapter The Python Profilers for more information on the Python profiler. The system’s profile function is called similarly to the system’s trace function (see settrace()), but it isn’t called for each executed line of code (only on call and return, but the return event is reported even when an exception has been set). The function is thread-specific, but there is no way for the profiler to know about context switches between threads, so it does not make sense to use this in the presence of multiple threads. Also, its return value is not used, so it can simply return None.

sys.setrecursionlimit(limit)
Set the maximum depth of the Python interpreter stack to limit. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.
The highest possible limit is platform-dependent. A user may need to set the limit higher when she has a program that requires deep recursion and a platform that supports a higher limit. This should be done with care, because a too-high limit can lead to a crash.

```
import sys
sys.settrace(tracefunc)
```

Set the system's trace function, which allows you to implement a Python source code debugger in Python. The function is thread-specific; for a debugger to support multiple threads, it must be registered using `sys.settrace()` for each thread being debugged.

Trace functions should have three arguments: `frame`, `event`, and `arg`. `frame` is the current stack frame. `event` is a string: 'call', 'line', 'return', 'exception', 'c_call', 'c_return', or 'c_exception'. `arg` depends on the event type.

The trace function is invoked (with `event` set to 'call') whenever a new local scope is entered; it should return a reference to a local trace function to be used that scope, or `None` if the scope shouldn’t be traced.

The local trace function should return a reference to itself (or to another function for further tracing in that scope), or `None` to turn off tracing in that scope.

The events have the following meaning:

`'call'`

A function is called (or some other code block entered). The global trace function is called; `arg` is `None`; the return value specifies the local trace function.

`'line'`

The interpreter is about to execute a new line of code or re-
execute the condition of a loop. The local trace function is called; *arg* is *None*; the return value specifies the new local trace function. See [Objects/lnotab_notes.txt](#) for a detailed explanation of how this works.

```
'return'
```

A function (or other code block) is about to return. The local trace function is called; *arg* is the value that will be returned, or *None* if the event is caused by an exception being raised. The trace function's return value is ignored.

```
'exception'
```

An exception has occurred. The local trace function is called; *arg* is a tuple (exception, value, traceback); the return value specifies the new local trace function.

```
'c_call'
```

A C function is about to be called. This may be an extension function or a built-in. *arg* is the C function object.

```
'c_return'
```

A C function has returned. *arg* is the C function object.

```
'c_exception'
```

A C function has raised an exception. *arg* is the C function object.

Note that as an exception is propagated down the chain of callers, an 'exception' event is generated at each level.

For more information on code and frame objects, refer to [The standard type hierarchy](#).

**CPython implementation detail:** The `settrace()` function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus
sys.setscdump(on_flag)

Activate dumping of VM measurements using the Pentium timestamp counter, if `on_flag` is true. Deactivate these dumps if `on_flag` is off. The function is available only if Python was compiled with `--with-tsc`. To understand the output of this dump, read `Python/ceval.c` in the Python sources.

*New in version 2.4.*

**CPython implementation detail:** This function is intimately bound to CPython implementation details and thus not likely to be implemented elsewhere.

sys.stdin
sys.stdout
sys.stderr

File objects corresponding to the interpreter’s standard input, output and error streams. `stdin` is used for all interpreter input except for scripts but including calls to `input()` and `raw_input()`. `stdout` is used for the output of `print` and `expression` statements and for the prompts of `input()` and `raw_input()`. The interpreter’s own prompts and (almost all of) its error messages go to `stderr`. `stdout` and `stderr` needn’t be built-in file objects: any object is acceptable as long as it has a `write()` method that takes a string argument. (Changing these objects doesn’t affect the standard I/O streams of processes executed by `os.popen()`, `os.system()` or the `exec*()` family of functions in the `os` module.)

sys.__stdin__
sys.__stdout__
sys.__stderr__
These objects contain the original values of `stdin`, `stderr` and `stdout` at the start of the program. They are used during finalization, and could be useful to print to the actual standard stream no matter if the `sys.std*` object has been redirected.

It can also be used to restore the actual files to known working file objects in case they have been overwritten with a broken object. However, the preferred way to do this is to explicitly save the previous stream before replacing it, and restore the saved object.

`sys.subversion`

A triple (repo, branch, version) representing the Subversion information of the Python interpreter. `repo` is the name of the repository, `CPython`. `branch` is a string of one of the forms 'trunk', 'branches/name' or 'tags/name'. `version` is the output of `svnversion`, if the interpreter was built from a Subversion checkout; it contains the revision number (range) and possibly a trailing ‘M’ if there were local modifications. If the tree was exported (or `svnversion` was not available), it is the revision of `Include/patchlevel.h` if the branch is a tag. Otherwise, it is `None`.

New in version 2.5.

**Note:** Python is now developed using Mercurial. In recent Python 2.7 bugfix releases, `subversion` therefore contains placeholder information. It is removed in Python 3.3.

`sys.tracebacklimit`

When this variable is set to an integer value, it determines the maximum number of levels of traceback information printed when an unhandled exception occurs. The default is `1000`. When set to `0` or less, all traceback information is suppressed and only the exception type and value are printed.
sys. **version**

A string containing the version number of the Python interpreter plus additional information on the build number and compiler used. This string is displayed when the interactive interpreter is started. Do not extract version information out of it, rather, use **version_info** and the functions provided by the **platform** module.

sys. **api_version**

The C API version for this interpreter. Programmers may find this useful when debugging version conflicts between Python and extension modules.

*New in version 2.3.*

sys. **version_info**

A tuple containing the five components of the version number: **major**, **minor**, **micro**, **releaselevel**, and **serial**. All values except **releaselevel** are integers; the release level is 'alpha', 'beta', 'candidate', or 'final'. The **version_info** value corresponding to the Python version 2.0 is (2, 0, 0, 'final', 0). The components can also be accessed by name, so **sys.version_info[0]** is equivalent to **sys.version_info.major** and so on.

*New in version 2.0.*

*Changed in version 2.7*: Added named component attributes

sys. **warnoptions**

This is an implementation detail of the warnings framework; do not modify this value. Refer to the **warnings** module for more information on the warnings framework.

sys. **winver**

The version number used to form registry keys on Windows
platforms. This is stored as string resource 1000 in the Python DLL. The value is normally the first three characters of `version`. It is provided in the `sys` module for informational purposes; modifying this value has no effect on the registry keys used by Python. Availability: Windows.

**Citations**

28.2. sysconfig — Provide access to Python’s configuration information

*New in version 2.7.*

**Source code:** Lib/sysconfig.py

The `sysconfig` module provides access to Python’s configuration information like the list of installation paths and the configuration variables relevant for the current platform.
28.2.1. Configuration variables

A Python distribution contains a Makefile and a `pyconfig.h` header file that are necessary to build both the Python binary itself and third-party C extensions compiled using `distutils`.

`sysconfig` puts all variables found in these files in a dictionary that can be accessed using `get_config_vars()` or `get_config_var()`.

Notice that on Windows, it’s a much smaller set.

`sysconfig.get_config_vars(*args)`

With no arguments, return a dictionary of all configuration variables relevant for the current platform.

With arguments, return a list of values that result from looking up each argument in the configuration variable dictionary.

For each argument, if the value is not found, return `None`.

`sysconfig.get_config_var(name)`

Return the value of a single variable `name`. Equivalent to `get_config_vars().get(name)`.

If `name` is not found, return `None`.

Example of usage:

```python
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
```
28.2.2. Installation paths

Python uses an installation scheme that differs depending on the platform and on the installation options. These schemes are stored in `sysconfig` under unique identifiers based on the value returned by `os.name`.

Every new component that is installed using `distutils` or a Distutils-based system will follow the same scheme to copy its file in the right places.

Python currently supports seven schemes:

- `posix_prefix`: scheme for Posix platforms like Linux or Mac OS X. This is the default scheme used when Python or a component is installed.
- `posix_home`: scheme for Posix platforms used when a `home` option is used upon installation. This scheme is used when a component is installed through Distutils with a specific home prefix.
- `posix_user`: scheme for Posix platforms used when a component is installed through Distutils and the `user` option is used. This scheme defines paths located under the user home directory.
- `nt`: scheme for NT platforms like Windows.
- `nt_user`: scheme for NT platforms, when the `user` option is used.
- `os2`: scheme for OS/2 platforms.
- `os2_home`: scheme for OS/2 platforms, when the `user` option is used.

Each scheme is itself composed of a series of paths and each path has a unique identifier. Python currently uses eight paths:

- `stdlib`: directory containing the standard Python library files that
are not platform-specific.
- `platstdlib`: directory containing the standard Python library files that are platform-specific.
- `platlib`: directory for site-specific, platform-specific files.
- `purelib`: directory for site-specific, non-platform-specific files.
- `include`: directory for non-platform-specific header files.
- `platininclude`: directory for platform-specific header files.
- `scripts`: directory for script files.
- `data`: directory for data files.

`sysconfig` provides some functions to determine these paths.

`sysconfig.get_scheme_names()`
Return a tuple containing all schemes currently supported in `sysconfig`.

`sysconfig.get_path_names()`
Return a tuple containing all path names currently supported in `sysconfig`.

`sysconfig.get_path(name[, scheme[, vars[, expand]]])`  
Return an installation path corresponding to the path `name`, from the install scheme named `scheme`.

`name` has to be a value from the list returned by `get_path_names()`.

`sysconfig` stores installation paths corresponding to each path name, for each platform, with variables to be expanded. For instance the `stdlib` path for the `nt` scheme is: `{base}/Lib`.

`get_path()` will use the variables returned by `get_config_vars()` to expand the path. All variables have default values for each platform so one may call this function and get the default value.
If `scheme` is provided, it must be a value from the list returned by `get_scheme_names()`. Otherwise, the default scheme for the current platform is used.

If `vars` is provided, it must be a dictionary of variables that will update the dictionary return by `get_config_vars()`.

If `expand` is set to `False`, the path will not be expanded using the variables.

If `name` is not found, return `None`.

```python
sysconfig.get_paths([scheme[, vars[, expand]]])
```

Return a dictionary containing all installation paths corresponding to an installation scheme. See `get_path()` for more information.

If `scheme` is not provided, will use the default scheme for the current platform.

If `vars` is provided, it must be a dictionary of variables that will update the dictionary used to expand the paths.

If `expand` is set to false, the paths will not be expanded.

If `scheme` is not an existing scheme, `get_paths()` will raise a `KeyError`.
28.2.3. Other functions

sysconfig.\texttt{get\_python\_version()}

Return the \texttt{MAJOR.MINOR} Python version number as a string. Similar to \texttt{sys.version[:3]}.

sysconfig.\texttt{get\_platform()}

Return a string that identifies the current platform.

This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by \texttt{os.uname()}), although the exact information included depends on the OS; e.g. for IRIX the architecture isn’t particularly important (IRIX only runs on SGI hardware), but for Linux the kernel version isn’t particularly important.

Examples of returned values:

- linux-i586
- linux-alpha (?)
- solaris-2.6-sun4u
- irix-5.3
- irix64-6.2

Windows will return one of:

- win-amd64 (64bit Windows on AMD64 (aka x86_64, Intel64, EM64T, etc)
- win-ia64 (64bit Windows on Itanium)
- win32 (all others - specifically, \texttt{sys.platform} is returned)

Mac OS X can return:

- macosx-10.6-ppc
macosx-10.4-ppc64
macosx-10.3-i386
macosx-10.4-fat

For other non-POSIX platforms, currently just returns `sys.platform`.

`sysconfig.is_python_build()`  
Return `True` if the current Python installation was built from source.

`sysconfig.parse_config_h(fp[, vars])`  
Parse a `config.h`-style file.

`fp` is a file-like object pointing to the `config.h`-like file.

A dictionary containing name/value pairs is returned. If an optional dictionary is passed in as the second argument, it is used instead of a new dictionary, and updated with the values read in the file.

`sysconfig.get_config_h_filename()`  
Return the path of `pyconfig.h`.

`sysconfig.get_makefile_filename()`  
Return the path of `Makefile`. 
28.3. __builtin__ — Built-in objects

This module provides direct access to all ‘built-in’ identifiers of Python; for example, __builtin__.open is the full name for the built-in function open(). See Built-in Functions and Built-in Constants for documentation.

This module is not normally accessed explicitly by most applications, but can be useful in modules that provide objects with the same name as a built-in value, but in which the built-in of that name is also needed. For example, in a module that wants to implement an open() function that wraps the built-in open(), this module can be used directly:

```python
import __builtin__

def open(path):
    f = __builtin__.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to upper-case.'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

# ...
```

**CPython implementation detail:** Most modules have the name __builtins__ (note the 's') made available as part of their globals. The value of __builtins__ is normally either this module or the value of this module's __dict__ attribute. Since this is an implementation detail, it may not be used by alternate
implementations of Python.
28.4. future_builtins — Python 3 builtins

New in version 2.6.

This module provides functions that exist in 2.x, but have different behavior in Python 3, so they cannot be put into the 2.x builtins namespace.

Instead, if you want to write code compatible with Python 3 builtins, import them from this module, like this:

```python
from future_builtins import map, filter
... code using Python 3-style map and filter ...
```

The 2to3 tool that ports Python 2 code to Python 3 will recognize this usage and leave the new builtins alone.

**Note:** The Python 3 `print()` function is already in the builtins, but cannot be accessed from Python 2 code unless you use the appropriate future statement:

```python
from __future__ import print_function
```

Available builtins are:

```python
future_builtins.ascii(object)
```

Returns the same as `repr()`. In Python 3, `repr()` will return printable Unicode characters unescaped, while `ascii()` will always backslash-escape them. Using `future_builtins.ascii()` instead of `repr()` in 2.6 code makes it clear that you need a pure ASCII return value.
**future_builtins.** **filter**(*function*, *iterable*)

Works like `itertools.ifilter()`.

**future_builtins.** **hex**(*object*)

Works like the built-in `hex()`, but instead of `__hex__()` it will use the `__index__()` method on its argument to get an integer that is then converted to hexadecimal.

**future_builtins.** **map**(*function*, *iterable*, ...)  

Works like `itertools.imap()`.

**Note:** In Python 3, `map()` does not accept `None` for the function argument.

**future_builtins.** **oct**(*object*)

Works like the built-in `oct()`, but instead of `__oct__()` it will use the `__index__()` method on its argument to get an integer that is then converted to octal.

**future_builtins.** **zip**(*iterables*)

Works like `itertools.izip()`.
28.5. __main__ — Top-level script environment

This module represents the (otherwise anonymous) scope in which the interpreter's main program executes — commands read either from standard input, from a script file, or from an interactive prompt. It is this environment in which the idiomatic “conditional script” stanza causes a script to run:

```python
if __name__ == "__main__":
    main()
```
Warning messages are typically issued in situations where it is useful to alert the user of some condition in a program, where that condition (normally) doesn’t warrant raising an exception and terminating the program. For example, one might want to issue a warning when a program uses an obsolete module.

Python programmers issue warnings by calling the `warn()` function defined in this module. (C programmers use `PyErr_WarnEx()`; see Exception Handling for details).

Warning messages are normally written to `sys.stderr`, but their disposition can be changed flexibly, from ignoring all warnings to turning them into exceptions. The disposition of warnings can vary based on the warning category (see below), the text of the warning message, and the source location where it is issued. Repetitions of a particular warning for the same source location are typically suppressed.

There are two stages in warning control: first, each time a warning is issued, a determination is made whether a message should be issued or not; next, if a message is to be issued, it is formatted and printed using a user-settable hook.

The determination whether to issue a warning message is controlled by the warning filter, which is a sequence of matching rules and actions. Rules can be added to the filter by calling `filterwarnings()` and reset to its default state by calling `resetwarnings()`.
The printing of warning messages is done by calling `showwarning()`, which may be overridden; the default implementation of this function formats the message by calling `formatwarning()`, which is also available for use by custom implementations.

**See also:** `logging.captureWarnings()` allows you to handle all warnings with the standard logging infrastructure.
28.6.1. Warning Categories

There are a number of built-in exceptions that represent warning categories. This categorization is useful to be able to filter out groups of warnings. The following warnings category classes are currently defined:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warning</td>
<td>This is the base class of all warning category classes. It is a subclass of Exception.</td>
</tr>
<tr>
<td>UserWarning</td>
<td>The default category for <code>warn()</code>.</td>
</tr>
<tr>
<td>DeprecationWarning</td>
<td>Base category for warnings about deprecated features (ignored by default).</td>
</tr>
<tr>
<td>SyntaxWarning</td>
<td>Base category for warnings about dubious syntactic features.</td>
</tr>
<tr>
<td>RuntimeWarning</td>
<td>Base category for warnings about dubious runtime features.</td>
</tr>
<tr>
<td>FutureWarning</td>
<td>Base category for warnings about constructs that will change semantically in the future.</td>
</tr>
<tr>
<td>PendingDeprecationWarning</td>
<td>Base category for warnings about features that will be deprecated in the future (ignored by default).</td>
</tr>
<tr>
<td>ImportWarning</td>
<td>Base category for warnings triggered during the process of importing a module (ignored by default).</td>
</tr>
<tr>
<td>UnicodeWarning</td>
<td>Base category for warnings related to Unicode.</td>
</tr>
</tbody>
</table>

While these are technically built-in exceptions, they are documented here, because conceptually they belong to the warnings mechanism.
User code can define additional warning categories by subclassing one of the standard warning categories. A warning category must always be a subclass of the `Warning` class.

*Changed in version 2.7:* `DeprecationWarning` is ignored by default.
28.6.2. The Warnings Filter

The warnings filter controls whether warnings are ignored, displayed, or turned into errors (raising an exception).

Conceptually, the warnings filter maintains an ordered list of filter specifications; any specific warning is matched against each filter specification in the list in turn until a match is found; the match determines the disposition of the match. Each entry is a tuple of the form \((\text{action}, \text{message}, \text{category}, \text{module}, \text{lineno})\), where:

- \textit{action} is one of the following strings:

<table>
<thead>
<tr>
<th>Value</th>
<th>Disposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;error&quot;</td>
<td>turn matching warnings into exceptions</td>
</tr>
<tr>
<td>&quot;ignore&quot;</td>
<td>never print matching warnings</td>
</tr>
<tr>
<td>&quot;always&quot;</td>
<td>always print matching warnings</td>
</tr>
<tr>
<td>&quot;default&quot;</td>
<td>print the first occurrence of matching warnings for each location where the warning is issued</td>
</tr>
<tr>
<td>&quot;module&quot;</td>
<td>print the first occurrence of matching warnings for each module where the warning is issued</td>
</tr>
<tr>
<td>&quot;once&quot;</td>
<td>print only the first occurrence of matching warnings, regardless of location</td>
</tr>
</tbody>
</table>

- \textit{message} is a string containing a regular expression that the start of the warning message must match. The expression is compiled to always be case-insensitive.

- \textit{category} is a class (a subclass of \texttt{Warning}) of which the warning category must be a subclass in order to match.

- \textit{module} is a string containing a regular expression that the module name must match. The expression is compiled to be case-sensitive.
• \textit{lineno} is an integer that the line number where the warning occurred must match, or 0 to match all line numbers.

Since the \texttt{Warning} class is derived from the built-in \texttt{Exception} class, to turn a warning into an error we simply raise \texttt{category(message)}.

The warnings filter is initialized by \texttt{-w} options passed to the Python interpreter command line. The interpreter saves the arguments for all \texttt{-w} options without interpretation in \texttt{sys.warnoptions}; the \texttt{warnings} module parses these when it is first imported (invalid options are ignored, after printing a message to \texttt{sys.stderr}).

\textbf{28.6.2.1. Default Warning Filters}

By default, Python installs several warning filters, which can be overridden by the command-line options passed to \texttt{-w} and calls to \texttt{filterwarnings()}.

• \texttt{DeprecationWarning} and \texttt{PendingDeprecationWarning}, and \texttt{ImportWarning} are ignored.
• \texttt{BytesWarning} is ignored unless the \texttt{-b} option is given once or twice; in this case this warning is either printed (\texttt{-b}) or turned into an exception (\texttt{-bb}).
28.6.3. Temporarily Suppressing Warnings

If you are using code that you know will raise a warning, such as a deprecated function, but do not want to see the warning, then it is possible to suppress the warning using the \texttt{catch\_warnings} context manager:

\begin{verbatim}
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
\end{verbatim}

While within the context manager all warnings will simply be ignored. This allows you to use known-deprecated code without having to see the warning while not suppressing the warning for other code that might not be aware of its use of deprecated code. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the \texttt{catch\_warnings} context manager at the same time, the behavior is undefined.
28.6.4. Testing Warnings

To test warnings raised by code, use the `catch_warnings` context manager. With it you can temporarily mutate the warnings filter to facilitate your testing. For instance, do the following to capture all raised warnings to check:

```python
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # Cause all warnings to always be triggered.
    warnings.simplefilter("always")
    # Trigger a warning.
    fxn()
    # Verify some things
    assert len(w) == 1
    assert issubclass(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

One can also cause all warnings to be exceptions by using `error` instead of `always`. One thing to be aware of is that if a warning has already been raised because of a `once/default` rule, then no matter what filters are set the warning will not be seen again unless the warnings registry related to the warning has been cleared.

Once the context manager exits, the warnings filter is restored to its state when the context was entered. This prevents tests from changing the warnings filter in unexpected ways between tests and leading to indeterminate test results. The `showwarning()` function in the module is also restored to its original value. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the `catch_warnings` context manager at the same time, the behavior is undefined.
When testing multiple operations that raise the same kind of warning, it is important to test them in a manner that confirms each operation is raising a new warning (e.g. set warnings to be raised as exceptions and check the operations raise exceptions, check that the length of the warning list continues to increase after each operation, or else delete the previous entries from the warnings list before each new operation).
28.6.5. Updating Code For New Versions of Python

Warnings that are only of interest to the developer are ignored by default. As such you should make sure to test your code with typically ignored warnings made visible. You can do this from the command-line by passing `-wd` to the interpreter (this is shorthand for `-W default`). This enables default handling for all warnings, including those that are ignored by default. To change what action is taken for encountered warnings you simply change what argument is passed to `-W`, e.g. `-W error`. See the `-W` flag for more details on what is possible.

To programmatically do the same as `-wd`, use:

```python
warnings.simplefilter('default')
```

Make sure to execute this code as soon as possible. This prevents the registering of what warnings have been raised from unexpectedly influencing how future warnings are treated.

Having certain warnings ignored by default is done to prevent a user from seeing warnings that are only of interest to the developer. As you do not necessarily have control over what interpreter a user uses to run their code, it is possible that a new version of Python will be released between your release cycles. The new interpreter release could trigger new warnings in your code that were not there in an older interpreter, e.g. `DeprecationWarning` for a module that you are using. While you as a developer want to be notified that your code is using a deprecated module, to a user this information is essentially noise and provides no benefit to them.
28.6.6. Available Functions

```
warnings.warn(message[, category[, stacklevel]])
```

Issue a warning, or maybe ignore it or raise an exception. The `category` argument, if given, must be a warning category class (see above); it defaults to `UserWarning`. Alternatively `message` can be a `Warning` instance, in which case `category` will be ignored and `message.__class__` will be used. In this case the message text will be `str(message)`. This function raises an exception if the particular warning issued is changed into an error by the warnings filter see above. The `stacklevel` argument can be used by wrapper functions written in Python, like this:

```
def deprecation(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

This makes the warning refer to `deprecation()`'s caller, rather than to the source of `deprecation()` itself (since the latter would defeat the purpose of the warning message).

```
warnings.warnExplicit(message, category, filename, lineno[, module[, registry[, module_globals]]]])
```

This is a low-level interface to the functionality of `warn()`, passing in explicitly the message, category, filename and line number, and optionally the module name and the registry (which should be the `__warningregistry__` dictionary of the module). The module name defaults to the filename with `.py` stripped; if no registry is passed, the warning is never suppressed. `message` must be a string and `category` a subclass of `Warning` or `message` may be a `Warning` instance, in which case `category` will be ignored.
module_globals, if supplied, should be the global namespace in use by the code for which the warning is issued. (This argument is used to support displaying source for modules found in zipfiles or other non-filesystem import sources).

*Changed in version 2.5:* Added the module_globals parameter.

```
warnings.warnpy3k(message[, category[, stacklevel]])
```

Issue a warning related to Python 3.x deprecation. Warnings are only shown when Python is started with the -3 option. Like `warn()` `message` must be a string and `category` a subclass of `Warning`. `warnpy3k()` is using `DeprecationWarning` as default warning class.

*New in version 2.6.*

```
warnings.showwarning(message, category, filename, lineno[, file[, line]])
```

Write a warning to a file. The default implementation calls `formatwarning(message, category, filename, lineno, line)` and writes the resulting string to `file`, which defaults to `sys.stderr`. You may replace this function with an alternative implementation by assigning to `warnings.showwarning`. `line` is a line of source code to be included in the warning message; if `line` is not supplied, `showwarning()` will try to read the line specified by `filename` and `lineno`.

*Changed in version 2.7:* The `line` argument is required to be supported.

```
warnings.formatwarning(message, category, filename, lineno[, line])
```

Format a warning the standard way. This returns a string which may contain embedded newlines and ends in a newline. `line` is a
line of source code to be included in the warning message; if line is not supplied, `formatwarning()` will try to read the line specified by `filename` and `lineno`.

*Changed in version 2.6:* Added the `line` argument.

```python
warnings.filterwarnings(action[, message[, category[, module[, lineno[, append]]]]])
```

Insert an entry into the list of `warnings` filter specifications. The entry is inserted at the front by default; if `append` is true, it is inserted at the end. This checks the types of the arguments, compiles the `message` and `module` regular expressions, and inserts them as a tuple in the list of warnings filters. Entries closer to the front of the list override entries later in the list, if both match a particular warning. Omitted arguments default to a value that matches everything.

```python
warnings.simplefilter(action[, category[, lineno[, append]]])
```

Insert a simple entry into the list of `warnings` filter specifications. The meaning of the function parameters is as for `filterwarnings()`, but regular expressions are not needed as the filter inserted always matches any message in any module as long as the category and line number match.

```python
warnings.resetwarnings()
```

Reset the warnings filter. This discards the effect of all previous calls to `filterwarnings()`, including that of the `-W` command line options and calls to `simplefilter()`.
28.6.7. Available Context Managers

```python
class warnings.catch_warnings([*, record=False, module=None])
```

A context manager that copies and, upon exit, restores the warnings filter and the `showwarning()` function. If the `record` argument is `False` (the default) the context manager returns `None` on entry. If `record` is `True`, a list is returned that is progressively populated with objects as seen by a custom `showwarning()` function (which also suppresses output to `sys.stdout`). Each object in the list has attributes with the same names as the arguments to `showwarning()`.

The `module` argument takes a module that will be used instead of the module returned when you import `warnings` whose filter will be protected. This argument exists primarily for testing the `warnings` module itself.

**Note:** The `catch_warnings` manager works by replacing and then later restoring the module's `showwarning()` function and internal list of filter specifications. This means the context manager is modifying global state and therefore is not thread-safe.

**Note:** In Python 3, the arguments to the constructor for `catch_warnings` are keyword-only arguments.

*New in version 2.6.*
28.7. contextlib — Utilities for with-statement contexts

*New in version 2.5.*

**Source code:** Lib/contextlib.py

This module provides utilities for common tasks involving the `with` statement. For more information see also Context Manager Types and With Statement Context Managers.

**Functions provided:**

```python
contextlib.contextmanager(func)
```

This function is a decorator that can be used to define a factory function for `with` statement context managers, without needing to create a class or separate `__enter__()` and `__exit__()` methods.

A simple example (this is not recommended as a real way of generating HTML!):

```
from contextlib import contextmanager

@contextmanager
def tag(name):
    print "<%s>" % name
    yield
    print "</%s>" % name

>>> with tag("h1"):
...    print "foo"
...
<h1>foo</h1>
```
The function being decorated must return a generator-iterator when called. This iterator must yield exactly one value, which will be bound to the targets in the with statement's as clause, if any.

At the point where the generator yields, the block nested in the with statement is executed. The generator is then resumed after the block is exited. If an unhandled exception occurs in the block, it is reraised inside the generator at the point where the yield occurred. Thus, you can use a try...except...finally statement to trap the error (if any), or ensure that some cleanup takes place. If an exception is trapped merely in order to log it or to perform some action (rather than to suppress it entirely), the generator must reraise that exception. Otherwise the generator context manager will indicate to the with statement that the exception has been handled, and execution will resume with the statement immediately following the with statement.

```
contextlib.nested(mgr1[, mgr2[, ...]])
```

Combine multiple context managers into a single nested context manager.

This function has been deprecated in favour of the multiple manager form of the with statement.

The one advantage of this function over the multiple manager form of the with statement is that argument unpacking allows it to be used with a variable number of context managers as follows:

```
from contextlib import nested

with nested(*managers):
    do_something()
```

Note that if the __exit__() method of one of the nested context managers indicates an exception should be suppressed, no
exception information will be passed to any remaining outer context managers. Similarly, if the \_exit\_() method of one of the nested managers raises an exception, any previous exception state will be lost; the new exception will be passed to the \_exit\_() methods of any remaining outer context managers. In general, \_exit\_() methods should avoid raising exceptions, and in particular they should not re-raise a passed-in exception.

This function has two major quirks that have led to it being deprecated. Firstly, as the context managers are all constructed before the function is invoked, the \_new\_() and \_init\_() methods of the inner context managers are not actually covered by the scope of the outer context managers. That means, for example, that using nested() to open two files is a programming error as the first file will not be closed promptly if an exception is thrown when opening the second file.

Secondly, if the \_enter\_() method of one of the inner context managers raises an exception that is caught and suppressed by the \_exit\_() method of one of the outer context managers, this construct will raise RuntimeError rather than skipping the body of the with statement.

Developers that need to support nesting of a variable number of context managers can either use the warnings module to suppress the DeprecationWarning raised by this function or else use this function as a model for an application specific implementation.

**Deprecated since version 2.7:** The with-statement now supports this functionality directly (without the confusing error prone quirks).
contextlib.closing(thing)

Return a context manager that closes thing upon completion of the block. This is basically equivalent to:

```python
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

And lets you write code like this:

```python
from contextlib import closing
import urllib

with closing(urllib.urlopen('http://www.python.org')) as page:
    for line in page:
        print line
```

without needing to explicitly close page. Even if an error occurs, page.close() will be called when the with block is exited.

See also:

PEP 343 - The “with” statement

The specification, background, and examples for the Python with statement.
New in version 2.6.

Source code: Lib/abc.py

This module provides the infrastructure for defining abstract base classes (ABCs) in Python, as outlined in PEP 3119; see the PEP for why this was added to Python. (See also PEP 3141 and the numbers module regarding a type hierarchy for numbers based on ABCs.)

The collections module has some concrete classes that derive from ABCs; these can, of course, be further derived. In addition the collections module has some ABCs that can be used to test whether a class or instance provides a particular interface, for example, is it hashable or a mapping.

This module provides the following class:

class abc. ABCMeta

Metaclass for defining Abstract Base Classes (ABCs).

Use this metaclass to create an ABC. An ABC can be subclassed directly, and then acts as a mix-in class. You can also register unrelated concrete classes (even built-in classes) and unrelated ABCs as “virtual subclasses” – these and their descendants will be considered subclasses of the registering ABC by the built-in issubclass() function, but the registering ABC won’t show up in their MRO (Method Resolution Order) nor will method implementations defined by the registering ABC be callable (not even via super()). [1]

Classes created with a metaclass of ABCMeta have the following
method:

**register(subclass)**

Register *subclass* as a “virtual subclass” of this ABC. For example:

```python
from abc import ABCMeta
class MyABC:
    __metaclass__ = ABCMeta
MyABC.register(tuple)
assertissubclass(tuple, MyABC)
assertissubclass((), MyABC)
```

You can also override this method in an abstract base class:

**__subclasshook__(subclass)**

(Must be defined as a class method.)

Check whether *subclass* is considered a subclass of this ABC. This means that you can customize the behavior of `issubclass` further without the need to call `register()` on every class you want to consider a subclass of the ABC. (This class method is called from the **__subclasscheck__()** method of the ABC.)

This method should return **True, False** or **NotImplemented**. If it returns **True**, the *subclass* is considered a subclass of this ABC. If it returns **False**, the *subclass* is not considered a subclass of this ABC, even if it would normally be one. If it returns **NotImplemented**, the subclass check is continued with the usual mechanism.

For a demonstration of these concepts, look at this example ABC definition:
The ABC `MyIterable` defines the standard iterable method, `__iter__()`, as an abstract method. The implementation given here can still be called from subclasses. The `get_iterator()` method is also part of the `MyIterable` abstract base class, but it does not have to be overridden in non-abstract derived classes.

The `__subclasshook__()` class method defined here says that any class that has an `__iter__()` method in its `__dict__` (or in that of one of its base classes, accessed via the `__mro__` list) is considered a `MyIterable` too.

Finally, the last line makes `Foo` a virtual subclass of `MyIterable`,
even though it does not define an \_\texttt{iter\_}(()) method (it uses the old-style iterable protocol, defined in terms of \_\texttt{len\_}(()) and \_\texttt{getitem\_}(())). Note that this will not make \texttt{get\_iterator} available as a method of \texttt{Foo}, so it is provided separately.

It also provides the following decorators:

\begin{description}
\item [abc.\texttt{abstractmethod}(\textit{function})] A decorator indicating abstract methods.
\end{description}

Using this decorator requires that the class’s metaclass is \texttt{ABCMeta} or is derived from it. A class that has a metaclass derived from \texttt{ABCMeta} cannot be instantiated unless all of its abstract methods and properties are overridden. The abstract methods can be called using any of the normal ‘super’ call mechanisms.

Dynamically adding abstract methods to a class, or attempting to modify the abstraction status of a method or class once it is created, are not supported. The \texttt{abstractmethod()} only affects subclasses derived using regular inheritance; “virtual subclasses” registered with the ABC’s \texttt{register()} method are not affected.

Usage:

\begin{python}
class \texttt{C}:
    \_\texttt{metaclass\_} = ABCMeta
@abstractmethod
    def \texttt{my\_abstract\_method}(\texttt{self}, \ldots):
        ...
\end{python}

\textbf{Note:} Unlike Java abstract methods, these abstract methods may have an implementation. This implementation can be called via the \texttt{super()} mechanism from the class that overrides it. This could be useful as an end-point for a super-call in a framework that uses cooperative multiple-inheritance.
abc.abstractproperty([fget[, fset[, fdel[, doc]]]])

A subclass of the built-in property(), indicating an abstract property.

Using this function requires that the class’s metaclass is ABCMeta or is derived from it. A class that has a metaclass derived from ABCMeta cannot be instantiated unless all of its abstract methods and properties are overridden. The abstract properties can be called using any of the normal ‘super’ call mechanisms.

Usage:

class C:
    __metaclass__ = ABCMeta
    @abstractproperty
def my_abstract_property(self):
        ...

This defines a read-only property; you can also define a read-write abstract property using the ‘long’ form of property declaration:

class C:
    __metaclass__ = ABCMeta
def getx(self): ...
def setx(self, value): ...
x = abstractproperty(getx, setx)

Footnotes

[1] C++ programmers should note that Python’s virtual base class concept is not the same as C++’s.
28.9. `atexit` — Exit handlers

New in version 2.0.

**Source code:** Lib/atexit.py

The `atexit` module defines a single function to register cleanup functions. Functions thus registered are automatically executed upon normal interpreter termination. `atexit` runs these functions in the reverse order in which they were registered; if you register `A`, `B`, and `C`, at interpreter termination time they will be run in the order `C`, `B`, `A`.

**Note:** The functions registered via this module are not called when the program is killed by a signal not handled by Python, when a Python fatal internal error is detected, or when `os._exit()` is called.

This is an alternate interface to the functionality provided by the `sys.exitfunc()` variable.

**Note:** This module is unlikely to work correctly when used with other code that sets `sys.exitfunc`. In particular, other core Python modules are free to use `atexit` without the programmer's knowledge. Authors who use `sys.exitfunc` should convert their code to use `atexit` instead. The simplest way to convert code that sets `sys.exitfunc` is to import `atexit` and register the function that had been bound to `sys.exitfunc`.

```python
atexit.register(func[, *args[, **kargs]])
```

Register `func` as a function to be executed at termination. Any optional arguments that are to be passed to `func` must be passed as arguments to `register()`. It is possible to register the same function and arguments more than once.
At normal program termination (for instance, if `sys.exit()` is called or the main module's execution completes), all functions registered are called in last in, first out order. The assumption is that lower level modules will normally be imported before higher level modules and thus must be cleaned up later.

If an exception is raised during execution of the exit handlers, a traceback is printed (unless `SystemExit` is raised) and the exception information is saved. After all exit handlers have had a chance to run the last exception to be raised is re-raised.

*Changed in version 2.6:* This function now returns `func`, which makes it possible to use it as a decorator.

---

**See also:**

**Module** `readline`

Useful example of `atexit` to read and write `readline` history files.
28.9.1. atexit Example

The following simple example demonstrates how a module can initialize a counter from a file when it is imported and save the counter's updated value automatically when the program terminates without relying on the application making an explicit call into this module at termination.

```python
try:
    _count = int(open("counter").read())
except IOError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    open("counter", "w").write("%d" % _count)

import atexit
atexit.register(savecounter)
```

Positional and keyword arguments may also be passed to `register()` to be passed along to the registered function when it is called:

```python
def goodbye(name, adjective):
    print 'Goodbye, %s, it was %s to meet you.' % (name, adjective)

import atexit
atexit.register(goodbye, 'Donny', 'nice')

# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```

Usage as a `decorator`: 
import atexit

@atexit.register
def goodbye():
    print "You are now leaving the Python sector."

This only works with functions that can be called without arguments.
This module provides a standard interface to extract, format and print stack traces of Python programs. It exactly mimics the behavior of the Python interpreter when it prints a stack trace. This is useful when you want to print stack traces under program control, such as in a “wrapper” around the interpreter.

The module uses traceback objects — this is the object type that is stored in the variables `sys.exc_traceback` (deprecated) and `sys.last_traceback` and returned as the third item from `sys.exc_info()`.

The module defines the following functions:

`traceback.print_tb(tb[, limit[, file]])`
Print up to `limit` stack trace entries from the traceback object `tb`. If `limit` is omitted or `None`, all entries are printed. If `file` is omitted or `None`, the output goes to `sys.stderr`; otherwise it should be an open file or file-like object to receive the output.

`traceback.print_exception(etype, value, tb[, limit[, file]])`
Print exception information and up to `limit` stack trace entries from the traceback `tb` to `file`. This differs from `print_tb()` in the following ways: (1) if `tb` is not `None`, it prints a header `Traceback (most recent call last)`:; (2) it prints the exception `etype` and `value` after the stack trace; (3) if `etype` is `SyntaxError` and `value` has the appropriate format, it prints the line where the syntax error occurred with a caret indicating the approximate position of the error.
traceback.\texttt{print}._\texttt{exc}([\texttt{limit}[, \texttt{file}]]]

This is a shorthand for \texttt{print}._\texttt{exception(sys.exc_type, sys.exc_value, sys.exc_traceback, limit, file)}. (In fact, it uses \texttt{sys.exc_info()} to retrieve the same information in a thread-safe way instead of using the deprecated variables.)

traceback.\texttt{format}._\texttt{exc}([\texttt{limit}])

This is like \texttt{print}._\texttt{exc(limit)} but returns a string instead of printing to a file.

\textit{New in version 2.4.}

traceback.\texttt{print}._\texttt{last}([\texttt{limit}[, \texttt{file}]]]

This is a shorthand for \texttt{print}._\texttt{exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file)}. In general it will work only after an exception has reached an interactive prompt (see \texttt{sys.last_type}).

traceback.\texttt{print}._\texttt{stack}([\texttt{f}[, \texttt{limit}[, \texttt{file}]]]

This function prints a stack trace from its invocation point. The optional \texttt{f} argument can be used to specify an alternate stack frame to start. The optional \texttt{limit} and \texttt{file} arguments have the same meaning as for \texttt{print}._\texttt{exception}.

traceback.\texttt{extract}._\texttt{tb}(\texttt{tb}[, \texttt{limit}])

Return a list of up to \texttt{limit} “pre-processed” stack trace entries extracted from the traceback object \texttt{tb}. It is useful for alternate formatting of stack traces. If \texttt{limit} is omitted or \texttt{None}, all entries are extracted. A “pre-processed” stack trace entry is a 4-tuple (\texttt{filename}, \texttt{line number}, function name*, \texttt{text}) representing the information that is usually printed for a stack trace. The \texttt{text} is a string with leading and trailing whitespace stripped; if the source
is not available it is None.

```
traceback.extract_stack([f, limit])
```

Extract the raw traceback from the current stack frame. The return value has the same format as for `extract_tb()`. The optional `f` and `limit` arguments have the same meaning as for `print_stack()`.

```
traceback.format_list(extracted_list)
```

Given a list of tuples as returned by `extract_tb()` or `extract_stack()`, return a list of strings ready for printing. Each string in the resulting list corresponds to the item with the same index in the argument list. Each string ends in a newline; the strings may contain internal newlines as well, for those items whose source text line is not None.

```
traceback.format_exception_only(etype, value)
```

Format the exception part of a traceback. The arguments are the exception type, `etype` and `value` such as given by `sys.last_type` and `sys.last_value`. The return value is a list of strings, each ending in a newline. Normally, the list contains a single string; however, for `SyntaxError` exceptions, it contains several lines that (when printed) display detailed information about where the syntax error occurred. The message indicating which exception occurred is the always last string in the list.

```
traceback.format_exception(etype, value, tb[[], limit])
```

Format a stack trace and the exception information. The arguments have the same meaning as the corresponding arguments to `print_exception()`. The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does `print_exception()`.
traceback.format_tb(tb[, limit])

A shorthand for \texttt{format_list(extract_tb(tb, limit))}.

traceback.format_stack([f[, limit]])

A shorthand for \texttt{format_list(extract_stack(f, limit))}.

traceback.tb_lineno(tb)

This function returns the current line number set in the traceback object. This function was necessary because in versions of Python prior to 2.3 when the \texttt{-o} flag was passed to Python the \texttt{tb.tb_lineno} was not updated correctly. This function has no use in versions past 2.3.
28.10.1. Traceback Examples

This simple example implements a basic read-eval-print loop, similar to (but less useful than) the standard Python interactive interpreter loop. For a more complete implementation of the interpreter loop, refer to the code module.

```python
import sys, traceback

def run_user_code(envdir):
    source = raw_input(">>> ")
    try:
        exec source in envdir
    except:
        print "Exception in user code:"
        print '-'*60
        traceback.print_exc(file=sys.stdout)
        print '-'*60

envdir = {}
while 1:
    run_user_code(envdir)
```

The following example demonstrates the different ways to print and format the exception and traceback:

```python
import sys, traceback

def lumberjack():
    bright_side_of_death()

def bright_side_of_death():
    return tuple()[0]

try:
    lumberjack()
except IndexError:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print "*** print_tb:"
    traceback.print_tb(exc_traceback, limit=1, file=sys.stdout)
    print "*** print_exception:"
The output for the example would look similar to this:

```python
print "*** print_tb:
    File "<doctest...>", line 10, in <module>
        lumberjack()
*** print_exception:
Traceback (most recent call last):
File "<doctest...>", line 10, in <module>
    lumberjack()
File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):
File "<doctest...>", line 10, in <module>
    lumberjack()
File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** format_exc, first and last line:
[Traceback (most recent call last):
    File "<doctest...>", line 10, in <module>
    lumberjack()
    File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
    File "<doctest...>", line 7, in bright_side_of_death
    return tuple()[0]
IndexError: tuple index out of range]
```
The following example shows the different ways to print and format the stack:

```python
>>> import traceback
>>> def another_function():
...    lumberstack()
...
>>> def lumberstack():
...    traceback.print_stack()
...    print repr(traceback.extract_stack())
...    print repr(traceback.format_stack())
...
>>> another_function()
  File "<doctest>", line 10, in another_function
  File "<doctest>", line 3, in another_function
      lumberstack()
  File "<doctest>", line 6, in lumberstack
      traceback.print_stack()

[('doctest...', 10, 'lumberjack()'),
 ('doctest...', 4, 'lumberjack', 'bright_side_of_death()'),
 ('doctest...', 7, 'bright_side_of_death', 'return tuple()[0]

This last example demonstrates the final few formatting functions:

```
```python
>>> an_error = IndexError('tuple index out of range')
>>> traceback.format_exception_only(type(an_error), an_error)
['IndexError: tuple index out of range
']
```
28.11. __future__ — Future statement definitions

Source code: Lib/__future__.py

__future__ is a real module, and serves three purposes:

- To avoid confusing existing tools that analyze import statements and expect to find the modules they’re importing.
- To ensure that future statements run under releases prior to 2.1 at least yield runtime exceptions (the import of __future__ will fail, because there was no module of that name prior to 2.1).
- To document when incompatible changes were introduced, and when they will be — or were — made mandatory. This is a form of executable documentation, and can be inspected programmatically via importing __future__ and examining its contents.

Each statement in __future__.py is of the form:

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease, CompilerFlag)
```

where, normally, OptionalRelease is less than MandatoryRelease, and both are 5-tuples of the same form as sys.version_info:

```
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
 PY_MINOR_VERSION, # the 1; an int
 PY_MICRO_VERSION, # the 0; an int
 PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; a string
 PY_RELEASE_SERIAL # the 3; an int
)
```
OptionalRelease records the first release in which the feature was accepted.

In the case of a MandatoryRelease that has not yet occurred, MandatoryRelease predicts the release in which the feature will become part of the language.

Else MandatoryRelease records when the feature became part of the language; in releases at or after that, modules no longer need a future statement to use the feature in question, but may continue to use such imports.

MandatoryRelease may also be None, meaning that a planned feature got dropped.

Instances of class _Feature have two corresponding methods, getOptionalRelease() and getMandatoryRelease().

CompilerFlag is the (bitfield) flag that should be passed in the fourth argument to the built-in function compile() to enable the feature in dynamically compiled code. This flag is stored in the compiler_flag attribute on _Feature instances.

No feature description will ever be deleted from __future__. Since its introduction in Python 2.1 the following features have found their way into the language using this mechanism:

<table>
<thead>
<tr>
<th>feature</th>
<th>optional in</th>
<th>mandatory in</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>nested_scopes</td>
<td>2.1.0b1</td>
<td>2.2</td>
<td><strong>PEP 227</strong>: Statically Nested Scopes</td>
</tr>
<tr>
<td>generators</td>
<td>2.2.0a1</td>
<td>2.3</td>
<td><strong>PEP 255</strong>: Simple Generators</td>
</tr>
<tr>
<td>division</td>
<td>2.2.0a2</td>
<td>3.0</td>
<td><strong>PEP 238</strong>: Changing the Division Operator</td>
</tr>
<tr>
<td>Feature</td>
<td>Version</td>
<td>Python Version</td>
<td>Notes</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------</td>
<td>----------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>absolute_import</td>
<td>2.5.0a1</td>
<td>3.0</td>
<td>PEP 328: Imports: Multi-Line and Absolute/Relative</td>
</tr>
<tr>
<td>with_statement</td>
<td>2.5.0a1</td>
<td>2.6</td>
<td>PEP 343: The “with” Statement</td>
</tr>
<tr>
<td>print_function</td>
<td>2.6.0a2</td>
<td>3.0</td>
<td>PEP 3105: Make print a function</td>
</tr>
<tr>
<td>unicode_literals</td>
<td>2.6.0a2</td>
<td>3.0</td>
<td>PEP 3112: Bytes literals in Python 3000</td>
</tr>
</tbody>
</table>

**See also:**

**Future statements**

How the compiler treats future imports.
28.12. gc — Garbage Collector Interface

This module provides an interface to the optional garbage collector. It provides the ability to disable the collector, tune the collection frequency, and set debugging options. It also provides access to unreachable objects that the collector found but cannot free. Since the collector supplements the reference counting already used in Python, you can disable the collector if you are sure your program does not create reference cycles. Automatic collection can be disabled by calling `gc.disable()`. To debug a leaking program call `gc.set_debug(gc.DEBUG_LEAK)`. Notice that this includes `gc.DEBUG_SAVEALL`, causing garbage-collected objects to be saved in `gc.garbage` for inspection.

The `gc` module provides the following functions:

`gc.enable()`  
Enable automatic garbage collection.

`gc.disable()`  
Disable automatic garbage collection.

`gc.isenabled()`  
Returns true if automatic collection is enabled.

`gc.collect([generation])`  
With no arguments, run a full collection. The optional argument `generation` may be an integer specifying which generation to collect (from 0 to 2). A `ValueError` is raised if the generation number is invalid. The number of unreachable objects found is returned.
*Changed in version 2.5:* The optional *generation* argument was added.

*Changed in version 2.6:* The free lists maintained for a number of built-in types are cleared whenever a full collection or collection of the highest generation (2) is run. Not all items in some free lists may be freed due to the particular implementation, in particular *int* and *float*.

```
> gc.set_debug(flags)
Set the garbage collection debugging flags. Debugging information will be written to *sys.stderr*. See below for a list of debugging flags which can be combined using bit operations to control debugging.

> gc.get_debug()
Return the debugging flags currently set.

> gc.get_objects()
Returns a list of all objects tracked by the collector, excluding the list returned.

*New in version 2.2.*

> gc.set_threshold(threshold0[, threshold1[, threshold2]])
Set the garbage collection thresholds (the collection frequency). Setting *threshold0* to zero disables collection.

The GC classifies objects into three generations depending on how many collection sweeps they have survived. New objects are placed in the youngest generation (generation 0). If an object survives a collection it is moved into the next older generation. Since generation 2 is the oldest generation, objects in that generation remain there after a collection. In order to decide when to run, the collector keeps track of the number object
allocations and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds \(threshold0\), collection starts. Initially only generation 0 is examined. If generation 0 has been examined more than \(threshold1\) times since generation 1 has been examined, then generation 1 is examined as well. Similarly, \(threshold2\) controls the number of collections of generation 1 before collecting generation 2.

\[
\text{gc.} \, \text{get\_count()}
\]

Return the current collection counts as a tuple of \((\text{count0}, \text{count1}, \text{count2})\).

*New in version 2.5.*

\[
\text{gc.} \, \text{get\_threshold()}
\]

Return the current collection thresholds as a tuple of \((\text{threshold0}, \text{threshold1}, \text{threshold2})\).

\[
\text{gc.} \, \text{get\_referrers(*objs)}
\]

Return the list of objects that directly refer to any of objs. This function will only locate those containers which support garbage collection; extension types which do refer to other objects but do not support garbage collection will not be found.

Note that objects which have already been dereferenced, but which live in cycles and have not yet been collected by the garbage collector can be listed among the resulting referrers. To get only currently live objects, call \text{collect()} before calling \text{get\_referrers()}.

Care must be taken when using objects returned by \text{get\_referrers()} because some of them could still be under construction and hence in a temporarily invalid state. Avoid using
get_referrers() for any purpose other than debugging.

New in version 2.2.

gc.get_referents(*objs)
Return a list of objects directly referred to by any of the arguments. The referents returned are those objects visited by the arguments' C-level tp_traverse methods (if any), and may not be all objects actually directly reachable. tp_traverse methods are supported only by objects that support garbage collection, and are only required to visit objects that may be involved in a cycle. So, for example, if an integer is directly reachable from an argument, that integer object may or may not appear in the result list.

New in version 2.3.

gc.is_tracked(obj)
Returns True if the object is currently tracked by the garbage collector, False otherwise. As a general rule, instances of atomic types aren't tracked and instances of non-atomic types (containers, user-defined objects...) are. However, some type-specific optimizations can be present in order to suppress the garbage collector footprint of simple instances (e.g. dicts containing only atomic keys and values):

```python
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
False
>>> gc.is_tracked({"a": []})
True
```
The following variable is provided for read-only access (you can mutate its value but should not rebind it):

\texttt{gc.garbage}

A list of objects which the collector found to be unreachable but could not be freed (uncollectable objects). By default, this list contains only objects with \texttt{__del__() methods}. [1] Objects that have \texttt{__del__() methods} and are part of a reference cycle cause the entire reference cycle to be uncollectable, including objects not necessarily in the cycle but reachable only from it. Python doesn’t collect such cycles automatically because, in general, it isn’t possible for Python to guess a safe order in which to run the \texttt{__del__() methods}. If you know a safe order, you can force the issue by examining the \texttt{garbage list}, and explicitly breaking cycles due to your objects within the list. Note that these objects are kept alive even so by virtue of being in the \texttt{garbage list}, so they should be removed from \texttt{garbage} too. For example, after breaking cycles, do \texttt{del gc.garbage[:]} to empty the list. It’s generally better to avoid the issue by not creating cycles containing objects with \texttt{__del__() methods}, and \texttt{garbage} can be examined in that case to verify that no such cycles are being created.

If \texttt{DEBUG_SAVEALL} is set, then all unreachable objects will be added to this list rather than freed.

The following constants are provided for use with \texttt{set_debug()}:

\texttt{gc.DEBUG_STATS}

Print statistics during collection. This information can be useful when tuning the collection frequency.
gc. **DEBUG_COLLECTABLE**
Print information on collectable objects found.

gc. **DEBUG_UNCOLLECTABLE**
Print information of uncollectable objects found (objects which are not reachable but cannot be freed by the collector). These objects will be added to the garbage list.

gc. **DEBUG_INSTANCES**
When **DEBUG_COLLECTABLE** or **DEBUG_UNCOLLECTABLE** is set, print information about instance objects found.

gc. **DEBUG_OBJECTS**
When **DEBUG_COLLECTABLE** or **DEBUG_UNCOLLECTABLE** is set, print information about objects other than instance objects found.

gc. **DEBUG_SAVEALL**
When set, all unreachable objects found will be appended to garbage rather than being freed. This can be useful for debugging a leaking program.

gc. **DEBUG_LEAK**
The debugging flags necessary for the collector to print information about a leaking program (equal to **DEBUG_COLLECTABLE** | **DEBUG_UNCOLLECTABLE** | **DEBUG_INSTANCES** | **DEBUG_OBJECTS** | **DEBUG_SAVEALL**).

Footnotes

[1] Prior to Python 2.2, the list contained all instance objects in unreachable cycles, not only those with `__del__()` methods.
28.13. inspect — Inspect live objects

New in version 2.1.

Source code: Lib/inspect.py

The inspect module provides several useful functions to help get information about live objects such as modules, classes, methods, functions, tracebacks, frame objects, and code objects. For example, it can help you examine the contents of a class, retrieve the source code of a method, extract and format the argument list for a function, or get all the information you need to display a detailed traceback.

There are four main kinds of services provided by this module: type checking, getting source code, inspecting classes and functions, and examining the interpreter stack.
28.13.1. Types and members

The `getmembers()` function retrieves the members of an object such as a class or module. The sixteen functions whose names begin with “is” are mainly provided as convenient choices for the second argument to `getmembers()`. They also help you determine when you can expect to find the following special attributes:

<table>
<thead>
<tr>
<th>Type</th>
<th>Attribute</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>module</td>
<td><strong>doc</strong></td>
<td>documentation string</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>file</strong></td>
<td>filename (missing for built-in modules)</td>
<td></td>
</tr>
<tr>
<td>class</td>
<td><strong>doc</strong></td>
<td>documentation string</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>module</strong></td>
<td>name of module in which this class was defined</td>
<td></td>
</tr>
<tr>
<td>method</td>
<td><strong>doc</strong></td>
<td>documentation string</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>name</strong></td>
<td>name with which this method was defined</td>
<td></td>
</tr>
<tr>
<td></td>
<td>im_class</td>
<td>class object that asked for this method</td>
<td>(1)</td>
</tr>
<tr>
<td></td>
<td>im_func or__func__</td>
<td>function object containing implementation of method</td>
<td></td>
</tr>
<tr>
<td></td>
<td>im_self or__self__</td>
<td>instance to which this method is bound, or None</td>
<td></td>
</tr>
<tr>
<td>function</td>
<td><strong>doc</strong></td>
<td>documentation string</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>name</strong></td>
<td>name with which this function was defined</td>
<td></td>
</tr>
<tr>
<td></td>
<td>func_code</td>
<td>code object containing compiled function bytecode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>func_defaults</td>
<td>tuple of any default values for arguments</td>
<td></td>
</tr>
<tr>
<td></td>
<td>func_doc</td>
<td>(same as <strong>doc</strong>)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>func_globals</td>
<td>global namespace in which</td>
<td></td>
</tr>
<tr>
<td><strong>generator</strong></td>
<td><strong>func_name</strong></td>
<td>this function was defined (same as <strong>name</strong>)</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td>---------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>iter</strong></td>
<td>close</td>
<td>defined to support iteration over container</td>
<td></td>
</tr>
<tr>
<td></td>
<td>close</td>
<td>raises new GeneratorExit exception inside the generator to terminate the iteration</td>
<td></td>
</tr>
<tr>
<td>gi_code</td>
<td>code object</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gi_frame</td>
<td>frame object or possibly None once the generator has been exhausted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gi_running</td>
<td>set to 1 when generator is executing, 0 otherwise</td>
<td></td>
<td></td>
</tr>
<tr>
<td>next</td>
<td>return the next item from the container</td>
<td></td>
<td></td>
</tr>
<tr>
<td>send</td>
<td>resumes the generator and &quot;sends&quot; a value that becomes the result of the current yield-expression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>throw</td>
<td>used to raise an exception inside the generator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>traceback</td>
<td>frame object at this level</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tb_frame</td>
<td>index of last attempted instruction in bytecode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tb_lasti</td>
<td>current line number in Python source code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tb_lineno</td>
<td>next inner traceback object (called by this level)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>frame</td>
<td>next outer frame object (this frame's caller)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_back</td>
<td>builtins namespace seen by this frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_builtins</td>
<td>code object being executed in this frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_code</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_exc_traceback</td>
<td>traceback if raised in this frame, or None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_exc_type</td>
<td>exception type if raised in this frame, or None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_exc_value</td>
<td>exception value if raised in this frame, or None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fGlobals</td>
<td>global namespace seen by this frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_lasti</td>
<td>index of last attempted instruction in bytecode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_lineno</td>
<td>current line number in Python source code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_locals</td>
<td>local namespace seen by this frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_restricted</td>
<td>0 or 1 if frame is in restricted execution mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_trace</td>
<td>tracing function for this frame, or None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_argcount</td>
<td>number of arguments (not including * or ** args)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_code</td>
<td>string of raw compiled bytecode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_consts</td>
<td>tuple of constants used in the bytecode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_filename</td>
<td>name of file in which this code object was created</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_firstlineno</td>
<td>number of first line in Python source code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_flags</td>
<td>bitmap: 1=optimized</td>
<td>2=newlocals</td>
<td>4=*arg</td>
</tr>
<tr>
<td>co_inotab</td>
<td>encoded mapping of line numbers to bytecode indices</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_name</td>
<td>name with which this code object was defined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_names</td>
<td>tuple of names of local variables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_nlocals</td>
<td>number of local variables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_stacksize</td>
<td>virtual machine stack space required</td>
<td></td>
<td></td>
</tr>
<tr>
<td>co_varnames</td>
<td>tuple of names of arguments and local variables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>builtin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>doc</strong></td>
<td>documentation string</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>name</strong></td>
<td>original name of this function or method</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>self</strong></td>
<td>instance to which a method is bound, or None</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note:

1. Changed in version 2.2: `im_class` used to refer to the class that defined the method.

**inspect.getmembers**(object[, predicate])

Return all the members of an object in a list of (name, value) pairs sorted by name. If the optional predicate argument is supplied, only members for which the predicate returns a true value are included.

**Note:**  `getmembers()` does not return metaclass attributes when the argument is a class (this behavior is inherited from the `dir()` function).

**inspect.getmoduleinfo**(path)

Return a tuple of values that describe how Python will interpret the file identified by path if it is a module, or None if it would not be identified as a module. The return tuple is (name, suffix, mode, module_type), where name is the name of the module without the name of any enclosing package, suffix is the trailing part of the
file name (which may not be a dot-delimited extension), mode is the open() mode that would be used ('r' or 'rb'), and module_type is an integer giving the type of the module. module_type will have a value which can be compared to the constants defined in the imp module; see the documentation for that module for more information on module types.

*Changed in version 2.6:* Returns a named tuple ModuleInfo(name, suffix, mode, module_type).

inspect.getmodulename(path)
Return the name of the module named by the file path, without including the names of enclosing packages. This uses the same algorithm as the interpreter uses when searching for modules. If the name cannot be matched according to the interpreter’s rules, None is returned.

inspect.ismodule(object)
Return true if the object is a module.

inspect.isclass(object)
Return true if the object is a class, whether built-in or created in Python code.

inspect.ismethod(object)
Return true if the object is a bound or unbound method written in Python.

inspect.isfunction(object)
Return true if the object is a Python function, which includes functions created by a lambda expression.

inspect.isgeneratorfunction(object)
Return true if the object is a Python generator function.
New in version 2.6.

inspect.**isgenerator**(object)
    Return true if the object is a generator.

New in version 2.6.

inspect.**istraceback**(object)
    Return true if the object is a traceback.

inspect.**isframe**(object)
    Return true if the object is a frame.

inspect.**iscode**(object)
    Return true if the object is a code.

inspect.**isbuiltin**(object)
    Return true if the object is a built-in function or a bound built-in method.

inspect.**isroutine**(object)
    Return true if the object is a user-defined or built-in function or method.

inspect.**isabstract**(object)
    Return true if the object is an abstract base class.

New in version 2.6.

inspect.**ismethoddescriptor**(object)
    Return true if the object is a method descriptor, but not if
    `ismethod()`, `isclass()`, `isfunction()` or `isbuiltin()` are true.

This is new as of Python 2.2, and, for example, is true of
`int.__add__`. An object passing this test has a `__get__()` method
but not a `__set__()` method, but beyond that the set of attributes
A `__name__` attribute is usually sensible, and `__doc__` often is.

Methods implemented via descriptors that also pass one of the other tests return false from the `ismethoddescriptor()` test, simply because the other tests promise more – you can, e.g., count on having the `im_func` attribute (etc) when an object passes `ismethod()`.

`inspect.isdatadescriptor(object)`
Return true if the object is a data descriptor.

Data descriptors have both a `__get__` and a `__set__` method. Examples are properties (defined in Python), getsets, and members. The latter two are defined in C and there are more specific tests available for those types, which is robust across Python implementations. Typically, data descriptors will also have `__name__` and `__doc__` attributes (properties, getsets, and members have both of these attributes), but this is not guaranteed.

_New in version 2.3._

`inspect.isgetsetdescriptor(object)`
Return true if the object is a getset descriptor.

_CPython implementation detail:_ getsets are attributes defined in extension modules via `PyGetSetDef` structures. For Python implementations without such types, this method will always return `False`.

_New in version 2.5._

`inspect.ismemberdescriptor(object)`
Return true if the object is a member descriptor.

**CPython implementation detail:** Member descriptors are attributes defined in extension modules via `PyMemberDef` structures. For Python implementations without such types, this method will always return `False`.

*New in version 2.5.*
28.13.2. Retrieving source code

**inspect.getdoc**(object)
Get the documentation string for an object, cleaned up with cleandoc().

**inspect.getcomments**(object)
Return in a single string any lines of comments immediately preceding the object’s source code (for a class, function, or method), or at the top of the Python source file (if the object is a module).

**inspect.getfile**(object)
Return the name of the (text or binary) file in which an object was defined. This will fail with a TypeError if the object is a built-in module, class, or function.

**inspect.getmodule**(object)
Try to guess which module an object was defined in.

**inspect.getsourcefile**(object)
Return the name of the Python source file in which an object was defined. This will fail with a TypeError if the object is a built-in module, class, or function.

**inspect.getsourcelines**(object)
Return a list of source lines and starting line number for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An IOError is raised if the source code cannot be retrieved.
**inspect.** `getsourc`e(*object*)

Return the text of the source code for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An `IOError` is raised if the source code cannot be retrieved.

**inspect.** `cleandoc`(*doc*)

Clean up indentation from docstrings that are indented to line up with blocks of code.

All leading whitespace is removed from the first line. Any leading whitespace that can be uniformly removed from the second line onwards is removed. Empty lines at the beginning and end are subsequently removed. Also, all tabs are expanded to spaces.

*New in version 2.6.*
28.13.3. Classes and functions

inspect.getclasstree(classes[, unique])

Arrange the given list of classes into a hierarchy of nested lists. Where a nested list appears, it contains classes derived from the class whose entry immediately precedes the list. Each entry is a 2-tuple containing a class and a tuple of its base classes. If the unique argument is true, exactly one entry appears in the returned structure for each class in the given list. Otherwise, classes using multiple inheritance and their descendants will appear multiple times.

inspect.getargspec(func)

Get the names and default values of a Python function's arguments. A tuple of four things is returned: (args, varargs, keywords, defaults). args is a list of the argument names (it may contain nested lists). varargs and keywords are the names of the * and ** arguments or None. defaults is a tuple of default argument values or None if there are no default arguments; if this tuple has n elements, they correspond to the last n elements listed in args.

Changed in version 2.6: Returns a named tuple ArgSpec(args, varargs, keywords, defaults).

inspect.getargvalues(frame)

Get information about arguments passed into a particular frame. A tuple of four things is returned: (args, varargs, keywords, locals). args is a list of the argument names (it may contain nested lists). varargs and keywords are the names of the * and ** arguments or None. locals is the locals dictionary of the given frame.
Changed in version 2.6: Returns a named tuple `ArgInfo(args, varargs, keywords, locals).

`inspect.formatargspec(args[, varargs, varkw, defaults, formatarg, formatvarargs, formatvarkw, formatvalue, join])`

Format a pretty argument spec from the four values returned by `getargspec()`. The format* arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

`inspect.formatargvalues(args[, varargs, varkw, locals, formatarg, formatvarargs, formatvarkw, formatvalue, join])`

Format a pretty argument spec from the four values returned by `getargvalues()`. The format* arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

`inspect.getmro(cls)`

Return a tuple of class cls’s base classes, including cls, in method resolution order. No class appears more than once in this tuple. Note that the method resolution order depends on cls’s type. Unless a very peculiar user-defined metatype is in use, cls will be the first element of the tuple.

`inspect.getcallargs(func[, *args][, **kwds])`

Bind the `args` and `kwds` to the argument names of the Python function or method `func`, as if it was called with them. For bound methods, bind also the first argument (typically named `self`) to the associated instance. A dict is returned, mapping the argument names (including the names of the * and ** arguments, if any) to their values from `args` and `kwds`. In case of invoking `func` incorrectly, i.e. whenever `func(*args, **kwds)` would raise an exception because of incompatible signature, an exception of the
same type and the same or similar message is raised. For example:

```python
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
>>> getcallargs(f, 1, 2, 3)
{'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
>>> getcallargs(f, a=2, x=4)
{'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
>>> getcallargs(f)
Traceback (most recent call last):
...  
TypeError: f() takes at least 1 argument (0 given)
```

*New in version 2.7.*
28.13.4. The interpreter stack

When the following functions return “frame records,” each record is a tuple of six items: the frame object, the filename, the line number of the current line, the function name, a list of lines of context from the source code, and the index of the current line within that list.

**Note:** Keeping references to frame objects, as found in the first element of the frame records these functions return, can cause your program to create reference cycles. Once a reference cycle has been created, the lifespan of all objects which can be accessed from the objects which form the cycle can become much longer even if Python’s optional cycle detector is enabled. If such cycles must be created, it is important to ensure they are explicitly broken to avoid the delayed destruction of objects and increased memory consumption which occurs.

Though the cycle detector will catch these, destruction of the frames (and local variables) can be made deterministic by removing the cycle in a `finally` clause. This is also important if the cycle detector was disabled when Python was compiled or using `gc.disable()`. For example:

```python
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

The optional `context` argument supported by most of these functions specifies the number of lines of context to return, which are centered around the current line.
inspect.getframeinfo(frame[, context])
Get information about a frame or traceback object. A 5-tuple is returned, the last five elements of the frame's frame record.

*Changed in version 2.6:* Returns a named tuple Traceback(filename, lineno, function, code_context, index).

inspect.getouterframes(frame[, context])
Get a list of frame records for a frame and all outer frames. These frames represent the calls that lead to the creation of frame. The first entry in the returned list represents frame; the last entry represents the outermost call on frame's stack.

inspect.getinnerframes(traceback[, context])
Get a list of frame records for a traceback's frame and all inner frames. These frames represent calls made as a consequence of frame. The first entry in the list represents traceback; the last entry represents where the exception was raised.

inspect.currentframe()
Return the frame object for the caller’s stack frame.

**CPython implementation detail:** This function relies on Python stack frame support in the interpreter, which isn’t guaranteed to exist in all implementations of Python. If running in an implementation without Python stack frame support this function returns None.

inspect.stack([context])
Return a list of frame records for the caller's stack. The first entry in the returned list represents the caller; the last entry represents the outermost call on the stack.
inspect.trace([context])

Return a list of frame records for the stack between the current frame and the frame in which an exception currently being handled was raised in. The first entry in the list represents the caller; the last entry represents where the exception was raised.
28.14. site — Site-specific configuration hook

Source code: Lib/site.py

This module is automatically imported during initialization. The automatic import can be suppressed using the interpreter's -s option.

Importing this module will append site-specific paths to the module search path and add a few builtins.

It starts by constructing up to four directories from a head and a tail part. For the head part, it uses sys.prefix and sys.exec_prefix; empty heads are skipped. For the tail part, it uses the empty string and then lib/site-packages (on Windows) or lib/pythonX.Y/site-packages and then lib/site-python (on Unix and Macintosh). For each of the distinct head-tail combinations, it sees if it refers to an existing directory, and if so, adds it to sys.path and also inspects the newly added path for configuration files.

A path configuration file is a file whose name has the form name.pth and exists in one of the four directories mentioned above; its contents are additional items (one per line) to be added to sys.path. Non-existing items are never added to sys.path, and no check is made that the item refers to a directory rather than a file. No item is added to sys.path more than once. Blank lines and lines beginning with # are skipped. Lines starting with import (followed by space or tab) are executed.

Changed in version 2.6: A space or tab is now required after the
import keyword.

For example, suppose `sys.prefix` and `sys.exec_prefix` are set to `/usr/local`. The Python X.Y library is then installed in `/usr/local/lib/pythonX.Y`. Suppose this has a subdirectory `/usr/local/lib/pythonX.Y/site-packages` with three subsubdirectories, `foo`, `bar` and `spam`, and two path configuration files, `foo.pth` and `bar.pth`. Assume `foo.pth` contains the following:

```python
# foo package configuration
foo
bar
bletch
```

and `bar.pth` contains:

```python
# bar package configuration
bar
```

Then the following version-specific directories are added to `sys.path`, in this order:

```python
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

Note that `bletch` is omitted because it doesn’t exist; the `bar` directory precedes the `foo` directory because `bar.pth` comes alphabetically before `foo.pth`; and `spam` is omitted because it is not mentioned in either path configuration file.

After these path manipulations, an attempt is made to import a module named `sitecustomize`, which can perform arbitrary site-specific customizations. It is typically created by a system administrator in the site-packages directory. If this import fails with an
**ImportError** exception, it is silently ignored. If Python is started without output streams available, as with `pythonw.exe` on Windows (which is used by default to start IDLE), attempted output from `sitecustomize` is ignored. Any exception other than **ImportError** causes a silent and perhaps mysterious failure of the process.

After this, an attempt is made to import a module named `usercustomize`, which can perform arbitrary user-specific customizations, if `ENABLE_USER_SITE` is true. This file is intended to be created in the user site-packages directory (see below), which is part of `sys.path` unless disabled by `-s`. An **ImportError** will be silently ignored.

Note that for some non-Unix systems, `sys.prefix` and `sys.exec_prefix` are empty, and the path manipulations are skipped; however the import of `sitecustomize` and `usercustomize` is still attempted.

```
site.PREFIXES
  A list of prefixes for site-packages directories.

  New in version 2.6.
```

```
site.ENABLE_USER_SITE
  Flag showing the status of the user site-packages directory. True means that it is enabled and was added to `sys.path`. False means that it was disabled by user request (with `-s` or `PYTHONNOUSERSITE`). None means it was disabled for security reasons (mismatch between user or group id and effective id) or by an administrator.

  New in version 2.6.
```

```
site.USER_SITE
Path to the user site-packages for the running Python. Can be None if `getusersitepackages()` hasn’t been called yet. Default value is `~/.local/lib/pythonX.Y/site-packages` for UNIX and non-framework Mac OS X builds, `~/Library/Python/X.Y/lib/python/site-packages` for Mac framework builds, and `%APPDATA%\Python\PythonXY\site-packages` on Windows. This directory is a site directory, which means that `.pth` files in it will be processed.

*New in version 2.6.*

**site.USER_BASE**

Path to the base directory for the user site-packages. Can be None if `getuserbase()` hasn’t been called yet. Default value is `~/.local` for UNIX and Mac OS X non-framework builds, `~/Library/Python/X.Y` for Mac framework builds, and `%APPDATA%\Python` for Windows. This value is used by Distutils to compute the installation directories for scripts, data files, Python modules, etc. for the user installation scheme. See also `PYTHONUSERBASE`.

*New in version 2.6.*

**site.addsitedir(sitedir, known_paths=None)**

Add a directory to sys.path and process its `.pth` files. Typically used in `sitecustomize` or `usercustomize` (see above).

**site.getsitepackages()**

Return a list containing all global site-packages directories (and possibly site-python).

*New in version 2.7.*

**site.getuserbase()**
Return the path of the user base directory, \texttt{USER\_BASE}. If it is not initialized yet, this function will also set it, respecting \texttt{PYTHONUSERBASE}.

\textit{New in version 2.7.}

\texttt{site.getusersitepackages()}

Return the path of the user-specific site-packages directory, \texttt{USER\_SITE}. If it is not initialized yet, this function will also set it, respecting \texttt{PYTHONNOUSERSITE} and \texttt{USER\_BASE}.

\textit{New in version 2.7.}

The \texttt{site} module also provides a way to get the user directories from the command line:

\begin{verbatim}
$ python -m site --user-site
/home/user/.local/lib/python2.7/site-packages
\end{verbatim}

If it is called without arguments, it will print the contents of \texttt{sys.path} on the standard output, followed by the value of \texttt{USER\_BASE} and whether the directory exists, then the same thing for \texttt{USER\_SITE}, and finally the value of \texttt{ENABLE\_USER\_SITE}.

\texttt{--user-base}

Print the path to the user base directory.

\texttt{--user-site}

Print the path to the user site-packages directory.

If both options are given, user base and user site will be printed (always in this order), separated by \texttt{os.pathsep}.

If any option is given, the script will exit with one of these values: \texttt{0} if the user site-packages directory is enabled, \texttt{1} if it was disabled by
the user, 2 if it is disabled for security reasons or by an administrator, and a value greater than 2 if there is an error.

See also: PEP 370 – Per user site-packages directory
28.15. user — User-specific configuration hook

*Deprecated since version 2.6:* The `user` module has been removed in Python 3.

As a policy, Python doesn't run user-specified code on startup of Python programs. (Only interactive sessions execute the script specified in the `PYTHONSTARTUP` environment variable if it exists).

However, some programs or sites may find it convenient to allow users to have a standard customization file, which gets run when a program requests it. This module implements such a mechanism. A program that wishes to use the mechanism must execute the statement

```
import user
```

The `user` module looks for a file `.pythonrc.py` in the user's home directory and if it can be opened, executes it (using `execfile()`) in its own (the module `user`'s) global namespace. Errors during this phase are not caught; that's up to the program that imports the `user` module, if it wishes. The home directory is assumed to be named by the `HOME` environment variable; if this is not set, the current directory is used.

The user's `.pythonrc.py` could conceivably test for `sys.version` if it wishes to do different things depending on the Python version.

A warning to users: be very conservative in what you place in your `.pythonrc.py` file. Since you don't know which programs will use it, changing the behavior of standard modules or functions is generally
not a good idea.

A suggestion for programmers who wish to use this mechanism: a simple way to let users specify options for your package is to have them define variables in their `.pythonrc.py` file that you test in your module. For example, a module `spam` that has a verbosity level can look for a variable `user.spam_verbose`, as follows:

```python
import user
verbose = bool(getattr(user, "spam_verbose", 0))
```

(The three-argument form of `getattr()` is used in case the user has not defined `spam_verbose` in their `.pythonrc.py` file.)

Programs with extensive customization needs are better off reading a program-specific customization file.

Programs with security or privacy concerns should not import this module; a user can easily break into a program by placing arbitrary code in the `.pythonrc.py` file.

Modules for general use should not import this module; it may interfere with the operation of the importing program.

**See also:**

**Module site**

Site-wide customization mechanism.
Most computers carry out floating point operations in conformance with the so-called IEEE-754 standard. On any real computer, some floating point operations produce results that cannot be expressed as a normal floating point value. For example, try

```python
>>> import math
>>> math.exp(1000)
inf
>>> math.exp(1000) / math.exp(1000)
nan
```

(The example above will work on many platforms. DEC Alpha may be one exception.) “Inf” is a special, non-numeric value in IEEE-754 that stands for “infinity”, and “nan” means “not a number.” Note that, other than the non-numeric results, nothing special happened when you asked Python to carry out those calculations. That is in fact the default behaviour prescribed in the IEEE-754 standard, and if it works for you, stop reading now.

In some circumstances, it would be better to raise an exception and stop processing at the point where the faulty operation was attempted. The `fpectl` module is for use in that situation. It provides control over floating point units from several hardware manufacturers, allowing the user to turn on the generation of `SIGFPE` whenever any of the IEEE-754 exceptions Division by Zero, Overflow, or Invalid Operation occurs. In tandem with a pair of
wrapper macros that are inserted into the C code comprising your python system, SIGFPE is trapped and converted into the Python FloatingPointError exception.

The fpectl module defines the following functions and may raise the given exception:

fpectl.turnon_sigfpe()
   Turn on the generation of SIGFPE, and set up an appropriate signal handler.

fpectl.turnoff_sigfpe()
   Reset default handling of floating point exceptions.

exception fpectl.FloatingPointError
   After turnon_sigfpe() has been executed, a floating point operation that raises one of the IEEE-754 exceptions Division by Zero, Overflow, or Invalid operation will in turn raise this standard Python exception.
28.16.1. Example

The following example demonstrates how to start up and test operation of the `fpectl` module.

```python
>>> import fpectl
>>> import fpetest
>>> fpectl.turnon_sigfpe()
>>> fpetest.test()
overflow    PASS
FloatingPointError: Overflow

div by 0    PASS
FloatingPointError: Division by zero

[ more output from test elided ]
```
28.16.2. Limitations and other considerations

Setting up a given processor to trap IEEE-754 floating point errors currently requires custom code on a per-architecture basis. You may have to modify `fpectl` to control your particular hardware.

Conversion of an IEEE-754 exception to a Python exception requires that the wrapper macros `PyFPE_START_PROTECT` and `PyFPE_END_PROTECT` be inserted into your code in an appropriate fashion. Python itself has been modified to support the `fpectl` module, but many other codes of interest to numerical analysts have not.

The `fpectl` module is not thread-safe.

**See also:** Some files in the source distribution may be interesting in learning more about how this module operates. The include file `Include/pyfpe.h` discusses the implementation of this module at some length. `Modules/fpetestmodule.c` gives several examples of use. Many additional examples can be found in `Objects/floatobject.c`. 
29. Custom Python Interpreters

The modules described in this chapter allow writing interfaces similar to Python's interactive interpreter. If you want a Python interpreter that supports some special feature in addition to the Python language, you should look at the code module. (The codeop module is lower-level, used to support compiling a possibly-incomplete chunk of Python code.)

The full list of modules described in this chapter is:

- 29.1. code — Interpreter base classes
  - 29.1.1. Interactive Interpreter Objects
  - 29.1.2. Interactive Console Objects
- 29.2. codeop — Compile Python code
29.1. code — Interpreter base classes

The code module provides facilities to implement read-eval-print loops in Python. Two classes and convenience functions are included which can be used to build applications which provide an interactive interpreter prompt.

```python
class code.InteractiveInterpreter([locals])
```

This class deals with parsing and interpreter state (the user's namespace); it does not deal with input buffering or prompting or input file naming (the filename is always passed in explicitly). The optional `locals` argument specifies the dictionary in which code will be executed; it defaults to a newly created dictionary with key '__name__' set to '__console__' and key '__doc__' set to None.

```python
class code.InteractiveConsole([locals[, filename]])
```

Closely emulate the behavior of the interactive Python interpreter. This class builds on InteractiveInterpreter and adds prompting using the familiar `sys.ps1` and `sys.ps2`, and input buffering.

```python
code.interact([banner[, readfunc[, local]]])
```

Convenience function to run a read-eval-print loop. This creates a new instance of InteractiveConsole and sets `readfunc` to be used as the InteractiveConsole.raw_input() method, if provided. If `local` is provided, it is passed to the InteractiveConsole constructor for use as the default namespace for the interpreter loop. The interact() method of the instance is then run with `banner` passed as the banner to use, if provided. The console object is discarded after use.
code.compile_command(source[, filename[, symbol]])

This function is useful for programs that want to emulate Python's interpreter main loop (a.k.a. the read-eval-print loop). The tricky part is to determine when the user has entered an incomplete command that can be completed by entering more text (as opposed to a complete command or a syntax error). This function *almost* always makes the same decision as the real interpreter main loop.

*source* is the source string; *filename* is the optional filename from which source was read, defaulting to 'input'; and *symbol* is the optional grammar start symbol, which should be either 'single' (the default) or 'eval'.

Returns a code object (the same as compile(source, filename, symbol)) if the command is complete and valid; None if the command is incomplete; raises **SyntaxError** if the command is complete and contains a syntax error, or raises **OverflowError** or **ValueError** if the command contains an invalid literal.
29.1.1. Interactive Interpreter Objects

InteractiveInterpreter.runsource(source[, filename[, symbol]])

Compile and run some source in the interpreter. Arguments are the same as for compile_command(); the default for filename is '<input>', and for symbol is 'single'. One several things can happen:

- The input is incorrect; compile_command() raised an exception (SyntaxError or OverflowError). A syntax traceback will be printed by calling the showsyntaxerror() method. runsource() returns False.
- The input is incomplete, and more input is required; compile_command() returned None. runsource() returns True.
- The input is complete; compile_command() returned a code object. The code is executed by calling the runcode() (which also handles run-time exceptions, except for SystemExit). runsource() returns False.

The return value can be used to decide whether to use sys.ps1 or sys.ps2 to prompt the next line.

InteractiveInterpreter.runcode(code)

Execute a code object. When an exception occurs, showtraceback() is called to display a traceback. All exceptions are caught except SystemExit, which is allowed to propagate.

A note about KeyboardInterrupt: this exception may occur elsewhere in this code, and may not always be caught. The caller should be prepared to deal with it.

InteractiveInterpreter.showsyntaxerror([filename])
Display the syntax error that just occurred. This does not display a stack trace because there isn’t one for syntax errors. If `filename` is given, it is stuffed into the exception instead of the default filename provided by Python’s parser, because it always uses ‘<string>’ when reading from a string. The output is written by the `write()` method.

`InteractiveInterpreter.showtraceback()`
Display the exception that just occurred. We remove the first stack item because it is within the interpreter object implementation. The output is written by the `write()` method.

`InteractiveInterpreter.write(data)`
Write a string to the standard error stream (`sys.stderr`). Derived classes should override this to provide the appropriate output handling as needed.
29.1.2. Interactive Console Objects

The `InteractiveConsole` class is a subclass of `InteractiveInterpreter`, and so offers all the methods of the interpreter objects as well as the following additions.

`InteractiveConsole.interact([banner])`

Closely emulate the interactive Python console. The optional banner argument specify the banner to print before the first interaction; by default it prints a banner similar to the one printed by the standard Python interpreter, followed by the class name of the console object in parentheses (so as not to confuse this with the real interpreter – since it's so close!).

`InteractiveConsole.push(line)`

Push a line of source text to the interpreter. The line should not have a trailing newline; it may have internal newlines. The line is appended to a buffer and the interpreter's `runsource()` method is called with the concatenated contents of the buffer as source. If this indicates that the command was executed or invalid, the buffer is reset; otherwise, the command is incomplete, and the buffer is left as it was after the line was appended. The return value is `True` if more input is required, `False` if the line was dealt with in some way (this is the same as `runsource()`).

`InteractiveConsole.resetbuffer()`

Remove any unhandled source text from the input buffer.

`InteractiveConsole.raw_input([prompt])`

Write a prompt and read a line. The returned line does not include the trailing newline. When the user enters the EOF key sequence, `EOFError` is raised. The base implementation uses the
built-in function `raw_input()`; a subclass may replace this with a different implementation.
29.2. codeop — Compile Python code

The codeop module provides utilities upon which the Python read-eval-print loop can be emulated, as is done in the code module. As a result, you probably don’t want to use the module directly; if you want to include such a loop in your program you probably want to use the code module instead.

There are two parts to this job:

1. Being able to tell if a line of input completes a Python statement: in short, telling whether to print ‘>>>’ or ‘...’ next.
2. Remembering which future statements the user has entered, so subsequent input can be compiled with these in effect.

The codeop module provides a way of doing each of these things, and a way of doing them both.

To do just the former:

```
codeop.compile_command(source[, filename[, symbol]])
```

Tries to compile source, which should be a string of Python code and return a code object if source is valid Python code. In that case, the filename attribute of the code object will be filename, which defaults to '<input>'. Returns None if source is not valid Python code, but is a prefix of valid Python code.

If there is a problem with source, an exception will be raised. SyntaxError is raised if there is invalid Python syntax, and OverflowError or ValueError if there is an invalid literal.

The symbol argument determines whether source is compiled as
a statement ("single", the default) or as an expression ("eval"). Any other value will cause ValueError to be raised.

**Note:** It is possible (but not likely) that the parser stops parsing with a successful outcome before reaching the end of the source; in this case, trailing symbols may be ignored instead of causing an error. For example, a backslash followed by two newlines may be followed by arbitrary garbage. This will be fixed once the API for the parser is better.

```python
class codeop.Compile
Instances of this class have __call__() methods identical in signature to the built-in function compile(), but with the difference that if the instance compiles program text containing a __future__ statement, the instance 'remembers' and compiles all subsequent program texts with the statement in force.

class codeop.CommandCompiler
Instances of this class have __call__() methods identical in signature to compile_command(); the difference is that if the instance compiles program text containing a __future__ statement, the instance 'remembers' and compiles all subsequent program texts with the statement in force.

A note on version compatibility: the Compile and CommandCompiler are new in Python 2.2. If you want to enable the future-tracking features of 2.2 but also retain compatibility with 2.1 and earlier versions of Python you can either write

```python
try:
    from codeop import CommandCompiler
    compile_command = CommandCompiler()
    del CommandCompiler
except ImportError:
    from codeop import compile_command
```
which is a low-impact change, but introduces possibly unwanted global state into your program, or you can write:

```python
try:
    from codeop import CommandCompiler
except ImportError:
    def CommandCompiler():
        from codeop import compile_command
        return compile_command
```

and then call `CommandCompiler` every time you need a fresh compiler object.
30. Restricted Execution

**Warning:** In Python 2.3 these modules have been disabled due to various known and not readily fixable security holes. The modules are still documented here to help in reading old code that uses the rexec and Bastion modules.

*Restricted execution* is the basic framework in Python that allows for the segregation of trusted and untrusted code. The framework is based on the notion that trusted Python code (a *supervisor*) can create a “padded cell’ (or environment) with limited permissions, and run the untrusted code within this cell. The untrusted code cannot break out of its cell, and can only interact with sensitive system resources through interfaces defined and managed by the trusted code. The term “restricted execution” is favored over “safe-Python” since true safety is hard to define, and is determined by the way the restricted environment is created. Note that the restricted environments can be nested, with inner cells creating subcells of lesser, but never greater, privilege.

An interesting aspect of Python’s restricted execution model is that the interfaces presented to untrusted code usually have the same names as those presented to trusted code. Therefore no special interfaces need to be learned to write code designed to run in a restricted environment. And because the exact nature of the padded cell is determined by the supervisor, different restrictions can be imposed, depending on the application. For example, it might be deemed “safe” for untrusted code to read any file within a specified directory, but never to write a file. In this case, the supervisor may redefine the built-in `open()` function so that it raises an exception whenever the *mode* parameter is `'w'`. It might also perform a `chroot()`-like operation on the *filename* parameter, such that root is always relative to some safe “sandbox” area of the filesystem. In this...
case, the untrusted code would still see a built-in `open()` function in its environment, with the same calling interface. The semantics would be identical too, with `IOError` being raised when the supervisor determined that an unallowable parameter is being used.

The Python run-time determines whether a particular code block is executing in restricted execution mode based on the identity of the `__builtins__` object in its global variables: if this is (the dictionary of) the standard `__builtin__` module, the code is deemed to be unrestricted, else it is deemed to be restricted.

Python code executing in restricted mode faces a number of limitations that are designed to prevent it from escaping from the padded cell. For instance, the function object attribute `func_globals` and the class and instance object attribute `__dict__` are unavailable.

Two modules provide the framework for setting up restricted execution environments:

- **30.1. rexec — Restricted execution framework**
  - **30.1.1. RExec Objects**
  - **30.1.2. Defining restricted environments**
  - **30.1.3. An example**
- **30.2. Bastion — Restricting access to objects**

**See also:**

**Grail Home Page**
Grail, an Internet browser written in Python, uses these modules to support Python applets. More information on the use of Python’s restricted execution mode in Grail is available on the Web site.
30.1. `rexec` — Restricted execution framework

Deprecation warning: The `rexec` module has been removed in Python 3.

Changed in version 2.3: Disabled module.

Warning: The documentation has been left in place to help in reading old code that uses the module.

This module contains the `RExec` class, which supports `r_eval()`, `r_execfile()`, `r_exec()`, and `r_import()` methods, which are restricted versions of the standard Python functions `eval()`, `execfile()` and the `exec` and `import` statements. Code executed in this restricted environment will only have access to modules and functions that are deemed safe; you can subclass `RExec` to add or remove capabilities as desired.

Warning: While the `rexec` module is designed to perform as described below, it does have a few known vulnerabilities which could be exploited by carefully written code. Thus it should not be relied upon in situations requiring "production ready" security. In such situations, execution via sub-processes or very careful "cleansing" of both code and data to be processed may be necessary. Alternatively, help in patching known `rexec` vulnerabilities would be welcomed.

Note: The `RExec` class can prevent code from performing unsafe operations like reading or writing disk files, or using TCP/IP.
sockets. However, it does not protect against code using extremely large amounts of memory or processor time.

```python
class rexec.RExec([hooks[, verbose]])
```

Returns an instance of the `RExec` class.

`hooks` is an instance of the `RHooks` class or a subclass of it. If it is omitted or `None`, the default `RHooks` class is instantiated. Whenever the `rexec` module searches for a module (even a built-in one) or reads a module's code, it doesn't actually go out to the file system itself. Rather, it calls methods of an `RHooks` instance that was passed to or created by its constructor. (Actually, the `RExec` object doesn’t make these calls — they are made by a module loader object that's part of the `RExec` object. This allows another level of flexibility, which can be useful when changing the mechanics of `import` within the restricted environment.)

By providing an alternate `RHooks` object, we can control the file system accesses made to import a module, without changing the actual algorithm that controls the order in which those accesses are made. For instance, we could substitute an `RHooks` object that passes all filesystem requests to a file server elsewhere, via some RPC mechanism such as ILU. Grail's applet loader uses this to support importing applets from a URL for a directory.

If `verbose` is true, additional debugging output may be sent to standard output.

It is important to be aware that code running in a restricted environment can still call the `sys.exit()` function. To disallow restricted code from exiting the interpreter, always protect calls that cause restricted code to run with a `try/except` statement that catches the `SystemExit` exception. Removing the `sys.exit()`
function from the restricted environment is not sufficient — the restricted code could still use `raise SystemExit`. Removing `SystemExit` is not a reasonable option; some library code makes use of this and would break were it not available.

**See also:**

**Grail Home Page**

Grail is a Web browser written entirely in Python. It uses the `reexec` module as a foundation for supporting Python applets, and can be used as an example usage of this module.
30.1.1. RExec Objects

RExec instances support the following methods:

**RExec. r_eval(code)**

*code* must either be a string containing a Python expression, or a compiled code object, which will be evaluated in the restricted environment's `__main__` module. The value of the expression or code object will be returned.

**RExec. r_exec(code)**

*code* must either be a string containing one or more lines of Python code, or a compiled code object, which will be executed in the restricted environment's `__main__` module.

**RExec. r_execfile(filename)**

Execute the Python code contained in the file *filename* in the restricted environment's `__main__` module.

Methods whose names begin with `s_` are similar to the functions beginning with `r_`, but the code will be granted access to restricted versions of the standard I/O streams `sys.stdin`, `sys.stderr`, and `sys.stdout`.

**RExec. s_eval(code)**

*code* must be a string containing a Python expression, which will be evaluated in the restricted environment.

**RExec. s_exec(code)**

*code* must be a string containing one or more lines of Python code, which will be executed in the restricted environment.

**RExec. s_execfile(code)**
Execute the Python code contained in the file `filename` in the restricted environment.

_RExec_ objects must also support various methods which will be implicitly called by code executing in the restricted environment. Overriding these methods in a subclass is used to change the policies enforced by a restricted environment.

```
RExec.r_import(modulename[, globals[, locals[, fromlist]]]])
```

Import the module `modulename`, raising an `ImportError` exception if the module is considered unsafe.

```
RExec.r_open(filename[, mode[, bufsize]])
```

Method called when `open()` is called in the restricted environment. The arguments are identical to those of `open()`, and a file object (or a class instance compatible with file objects) should be returned. _RExec_'s default behaviour is allow opening any file for reading, but forbidding any attempt to write a file. See the example below for an implementation of a less restrictive `r_open()`.

```
RExec.r_reload(module)
```

Reload the module object `module`, re-parsing and re-initializing it.

```
RExec.r_unload(module)
```

Unload the module object `module` (remove it from the restricted environment's `sys.modules` dictionary).

And their equivalents with access to restricted standard I/O streams:

```
RExec.s_import(modulename[, globals[, locals[, fromlist]]]])
```

Import the module `modulename`, raising an `ImportError` exception if the module is considered unsafe.
RExec.s_reload(module)
Reload the module object \textit{module}, re-parsing and re-initializing it.

RExec.s_unload(module)
Unload the module object \textit{module}. 
30.1.2. Defining restricted environments

The `RExec` class has the following class attributes, which are used by the `__init__()` method. Changing them on an existing instance won’t have any effect; instead, create a subclass of `RExec` and assign them new values in the class definition. Instances of the new class will then use those new values. All these attributes are tuples of strings.

`RExec.nok_builtin_names`

Contains the names of built-in functions which will not be available to programs running in the restricted environment. The value for `RExec` is `('open', 'reload', '__import__')`. (This gives the exceptions, because by far the majority of built-in functions are harmless. A subclass that wants to override this variable should probably start with the value from the base class and concatenate additional forbidden functions — when new dangerous built-in functions are added to Python, they will also be added to this module.)

`RExec.ok_builtin_modules`

Contains the names of built-in modules which can be safely imported. The value for `RExec` is `('audioop', 'array', 'binascii', 'cmath', 'errno', 'imageop', 'marshal', 'math', 'md5', 'operator', 'parser', 'regex', 'select', 'sha', '_sre', 'strop', 'struct', 'time')`. A similar remark about overriding this variable applies — use the value from the base class as a starting point.

`RExec.ok_path`

Contains the directories which will be searched when an `import` is performed in the restricted environment. The value for `RExec` is the same as `sys.path` (at the time the module is loaded) for
unrestricted code.

**RExec. ok_posix_names**

Contains the names of the functions in the *os* module which will be available to programs running in the restricted environment. The value for *RExec* is ('error', 'fstat', 'listdir', 'lstat', 'readlink', 'stat', 'times', 'uname', 'getpid', 'getppid', 'getcwd', 'getuid', 'getgid', 'geteuid', 'getegid').

**RExec. ok_sys_names**

Contains the names of the functions and variables in the *sys* module which will be available to programs running in the restricted environment. The value for *RExec* is ('ps1', 'ps2', 'copyright', 'version', 'platform', 'exit', 'maxint').

**RExec. ok_file_types**

Contains the file types from which modules are allowed to be loaded. Each file type is an integer constant defined in the *imp* module. The meaningful values are *PY_SOURCE*, *PY_COMPILED*, and *C_EXTENSION*. The value for *RExec* is (*C_EXTENSION*, *PY_SOURCE*). Adding *PY_COMPILED* in subclasses is not recommended; an attacker could exit the restricted execution mode by putting a forged byte-compiled file (*.pyc*) anywhere in your file system, for example by writing it to */tmp* or uploading it to the */incoming* directory of your public FTP server.
30.1.3. An example

Let us say that we want a slightly more relaxed policy than the standard \texttt{RExec} class. For example, if we're willing to allow files in \texttt{/tmp} to be written, we can subclass the \texttt{RExec} class:

```python
class TmpWriterRExec(rexec.RExec):
    def r_open(self, file, mode='r', buf=-1):
        if mode in ('r', 'rb'):
            pass
        elif mode in ('w', 'wb', 'a', 'ab'):
            # check filename: must begin with /tmp/
            if file[5:]!='/tmp/':
                raise IOError("can't write outside /tmp")
            elif string.find(file, '/../') >= 0 or
            file[3:] == '/../' or file[-3:] == '/../':
                raise IOError("'..' in filename forbidden")
        else:
            raise IOError("Illegal open() mode")
        return open(file, mode, buf)
```

Notice that the above code will occasionally forbid a perfectly valid filename; for example, code in the restricted environment won't be able to open a file called \texttt{/tmp/foo/../bar}. To fix this, the \texttt{r_open()} method would have to simplify the filename to \texttt{/tmp/bar}, which would require splitting apart the filename and performing various operations on it. In cases where security is at stake, it may be preferable to write simple code which is sometimes overly restrictive, instead of more general code that is also more complex and may harbor a subtle security hole.
30.2. Bastion — Restricting access to objects

*Deprecated since version 2.6:* The `Bastion` module has been removed in Python 3.

*Changed in version 2.3:* Disabled module.

**Note:** The documentation has been left in place to help in reading old code that uses the module.

According to the dictionary, a bastion is “a fortified area or position”, or “something that is considered a stronghold.” It’s a suitable name for this module, which provides a way to forbid access to certain attributes of an object. It must always be used with the `reexec` module, in order to allow restricted-mode programs access to certain safe attributes of an object, while denying access to other, unsafe attributes.

```
Bastion.Bastion(object[, filter[, name[, class]]])
```

Protect the object `object`, returning a bastion for the object. Any attempt to access one of the object’s attributes will have to be approved by the `filter` function; if the access is denied an `AttributeError` exception will be raised.

If present, `filter` must be a function that accepts a string containing an attribute name, and returns true if access to that attribute will be permitted; if `filter` returns false, the access is denied. The default filter denies access to any function beginning with an underscore (`'_'`). The bastion’s string representation will be `<Bastion for name>` if a value for `name` is provided; otherwise,
repr(object) will be used.

class, if present, should be a subclass of BastionClass; see the code in bastion.py for the details. Overriding the default BastionClass will rarely be required.

class Bastion. BastionClass(getfunc, name)
Class which actually implements bastion objects. This is the default class used by Bastion(). The getfunc parameter is a function which returns the value of an attribute which should be exposed to the restricted execution environment when called with the name of the attribute as the only parameter. name is used to construct the repr() of the BastionClass instance.
31. Importing Modules

The modules described in this chapter provide new ways to import other Python modules and hooks for customizing the import process.

The full list of modules described in this chapter is:

- 31.1. imp — Access the import internals
  - 31.1.1. Examples
- 31.2. importlib — Convenience wrappers for __import__()  
- 31.3. imputil — Import utilities  
  - 31.3.1. Examples
- 31.4. zipimport — Import modules from Zip archives  
  - 31.4.1. zipimporter Objects  
  - 31.4.2. Examples
- 31.5. pkgutil — Package extension utility
- 31.6. modulefinder — Find modules used by a script 
  - 31.6.1. Example usage of ModuleFinder
- 31.7. runpy — Locating and executing Python modules
31.1. imp — Access the import internals

This module provides an interface to the mechanisms used to implement the import statement. It defines the following constants and functions:

imp.get_magic()
Return the magic string value used to recognize byte-compiled code files (.pyc files). (This value may be different for each Python version.)

imp.get_suffixes()
Return a list of 3-element tuples, each describing a particular type of module. Each triple has the form (suffix, mode, type), where suffix is a string to be appended to the module name to form the filename to search for, mode is the mode string to pass to the built-in open() function to open the file (this can be 'r' for text files or 'rb' for binary files), and type is the file type, which has one of the values PY_SOURCE, PY_COMPILED, or C_EXTENSION, described below.

imp.find_module(name[, path])
Try to find the module name. If path is omitted or None, the list of directory names given by sys.path is searched, but first a few special places are searched: the function tries to find a built-in module with the given name (C_BUILTIN), then a frozen module (PY_FROZEN), and on some systems some other places are looked in as well (on Windows, it looks in the registry which may point to a specific file).

Otherwise, path must be a list of directory names; each directory
is searched for files with any of the suffixes returned by
get_suffixes() above. Invalid names in the list are silently
ignored (but all list items must be strings).

If search is successful, the return value is a 3-element tuple
(file, pathname, description):

file is an open file object positioned at the beginning, pathname is
the pathname of the file found, and description is a 3-element
tuple as contained in the list returned by get_suffixes()
describing the kind of module found.

If the module does not live in a file, the returned file is None,
pathname is the empty string, and the description tuple contains
empty strings for its suffix and mode; the module type is indicated
as given in parentheses above. If the search is unsuccessful,
ImportError is raised. Other exceptions indicate problems with
the arguments or environment.

If the module is a package, file is None, pathname is the package
path and the last item in the description tuple is PKG_DIRECTORY.

This function does not handle hierarchical module names (names
containing dots). In order to find P.M, that is, submodule M of
package P, use find_module() and load_module() to find and
load package P, and then use find_module() with the path
argument set to P.__path__. When P itself has a dotted name,
apply this recipe recursively.

```python
import load_module

load_module(name, file, pathname, description)
```

Load a module that was previously found by find_module() (or by
an otherwise conducted search yielding compatible results). This
function does more than importing the module: if the module was
already imported, it is equivalent to a reload()! The name
argument indicates the full module name (including the package name, if this is a submodule of a package). The file argument is an open file, and pathname is the corresponding file name; these can be None and '', respectively, when the module is a package or not being loaded from a file. The description argument is a tuple, as would be returned by get_suffixes(), describing what kind of module must be loaded.

If the load is successful, the return value is the module object; otherwise, an exception (usually ImportError) is raised.

**Important:** the caller is responsible for closing the file argument, if it was not None, even when an exception is raised. This is best done using a try ... finally statement.

```python
imp.new_module(name)
```

Return a new empty module object called name. This object is not inserted in sys.modules.

```python
imp.lock_held()
```

Return True if the import lock is currently held, else False. On platforms without threads, always return False.

On platforms with threads, a thread executing an import holds an internal lock until the import is complete. This lock blocks other threads from doing an import until the original import completes, which in turn prevents other threads from seeing incomplete module objects constructed by the original thread while in the process of completing its import (and the imports, if any, triggered by that).

```python
imp.acquire_lock()
```

Acquire the interpreter’s import lock for the current thread. This lock should be used by import hooks to ensure thread-safety
when importing modules.

Once a thread has acquired the import lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

On platforms without threads, this function does nothing.

*New in version 2.3.*

**imp.** `release_lock()`

Release the interpreter’s import lock. On platforms without threads, this function does nothing.

*New in version 2.3.*

The following constants with integer values, defined in this module, are used to indicate the search result of `find_module()`.

**imp.** `PY_SOURCE`

The module was found as a source file.

**imp.** `PY_COMPILED`

The module was found as a compiled code object file.

**imp.** `C_EXTENSION`

The module was found as a dynamically loadable shared library.

**imp.** `PKG_DIRECTORY`

The module was found as a package directory.

**imp.** `C_BUILTIN`

The module was found as a built-in module.

**imp.** `PY_FROZEN`

The module was found as a frozen module (see `init_frozen()`).
The following constant and functions are obsolete; their functionality is available through \texttt{find\_module()} or \texttt{load\_module()}. They are kept around for backward compatibility:

\texttt{imp.\nm SEARCH\_ERROR}

Unused.

\texttt{imp.\nin\_builtin(name)\n}

Initialize the built-in module called \textit{name} and return its module object along with storing it in \texttt{sys.modules}. If the module was already initialized, it will be initialized again. Re-initialization involves the copying of the built-in module's \texttt{\_dict\_} from the cached module over the module's entry in \texttt{sys.modules}. If there is no built-in module called \textit{name}, \texttt{None} is returned.

\texttt{imp.\nin\_frozen(name)\n}

Initialize the frozen module called \textit{name} and return its module object. If the module was already initialized, it will be initialized again. If there is no frozen module called \textit{name}, \texttt{None} is returned. (Frozen modules are modules written in Python whose compiled byte-code object is incorporated into a custom-built Python interpreter by Python's \texttt{freeze} utility. See \texttt{Tools/freeze/} for now.)

\texttt{imp.\nis\_builtin(name)\n}

Return 1 if there is a built-in module called \textit{name} which can be initialized again. Return -1 if there is a built-in module called \textit{name} which cannot be initialized again (see \texttt{init\_builtin()}). Return 0 if there is no built-in module called \textit{name}.

\texttt{imp.\nis\_frozen(name)\n}

Return \texttt{True} if there is a frozen module (see \texttt{init\_frozen()}) called \textit{name}, or \texttt{False} if there is no such module.

\texttt{imp.\nload\_compiled(name, pathname\[, file])\n}
Load and initialize a module implemented as a byte-compiled code file and return its module object. If the module was already initialized, it will be initialized again. The name argument is used to create or access a module object. The pathname argument points to the byte-compiled code file. The file argument is the byte-compiled code file, open for reading in binary mode, from the beginning. It must currently be a real file object, not a user-defined class emulating a file.

```python
imp.load_dynamic(name, pathname[, file])
```

Load and initialize a module implemented as a dynamically loadable shared library and return its module object. If the module was already initialized, it will be initialized again. Re-initialization involves copying the `__dict__` attribute of the cached instance of the module over the value used in the module cached in `sys.modules`. The pathname argument must point to the shared library. The name argument is used to construct the name of the initialization function: an external C function called `initname()` in the shared library is called. The optional file argument is ignored. (Note: using shared libraries is highly system dependent, and not all systems support it.)

**CPython implementation detail:** The import internals identify extension modules by filename, so doing `foo = load_dynamic("foo", "mod.so")` and `bar = load_dynamic("bar", "mod.so")` will result in both foo and bar referring to the same module, regardless of whether or not `mod.so` exports an `initbar` function. On systems which support them, symlinks can be used to import multiple modules from the same shared library, as each reference to the module will use a different file name.
imp. **load_source**(name, pathname[, file])

Load and initialize a module implemented as a Python source file and return its module object. If the module was already initialized, it will be initialized again. The *name* argument is used to create or access a module object. The *pathname* argument points to the source file. The *file* argument is the source file, open for reading as text, from the beginning. It must currently be a real file object, not a user-defined class emulating a file. Note that if a properly matching byte-compiled file (with suffix .pyc or .pyo) exists, it will be used instead of parsing the given source file.

**class imp. NullImporter(path_string)**

The **NullImporter** type is a PEP 302 import hook that handles non-directory path strings by failing to find any modules. Calling this type with an existing directory or empty string raises **ImportError**. Otherwise, a **NullImporter** instance is returned.

Python adds instances of this type to **sys.path_importer_cache** for any path entries that are not directories and are not handled by any other path hooks on **sys.path_hooks**. Instances have only one method:

**find_module**(fullname[, path])

This method always returns **None**, indicating that the requested module could not be found.

*New in version 2.5.*
31.1.1. Examples

The following function emulates what was the standard import statement up to Python 1.4 (no hierarchical module names). (This implementation wouldn't work in that version, since `find_module()` has been extended and `load_module()` has been added in 1.4.)

```python
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
    # Fast path: see if the module has already been imported.
    try:
        return sys.modules[name]
    except KeyError:
        pass

    # If any of the following calls raises an exception,
    # there's a problem we can't handle -- let the caller handle it.
    fp, pathname, description = imp.find_module(name)

    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        # Since we may exit via an exception, close fp explicitly.
        if fp:
            fp.close()
```

A more complete example that implements hierarchical module names and includes a `reload()` function can be found in the module `knee`. The `knee` module can be found in Demo/imputil/ in the Python source distribution.
31.2. importlib — Convenience wrappers for __import__()  

_New in version 2.7._

This module is a minor subset of what is available in the more full-featured package of the same name from Python 3.1 that provides a complete implementation of `import`. What is here has been provided to help ease in transitioning from 2.7 to 3.1.

```python
import import_module(name, package=None)
```

Import a module. The `name` argument specifies what module to import in absolute or relative terms (e.g. either `pkg.mod` or `.mod`). If the name is specified in relative terms, then the `package` argument must be specified to the package which is to act as the anchor for resolving the package name (e.g. `import_module('..mod', 'pkg.subpkg')` will import `pkg.mod`). The specified module will be inserted into `sys.modules` and returned.
31. Importing Modules

In this section, we will cover the basics of importing modules in Python. Modules are files that contain Python code that can be used by other programs or scripts. Modules are used to organize code into logical units, making it easier to manage and reuse code.

To import a module, you can use the `import` statement. For example:

```python
import math
```

This will import the `math` module, which contains a variety of mathematical functions.

You can also import specific functions or variables from a module using the `from` statement. For example:

```python
from math import pi
```

This will import the `pi` constant from the `math` module.

For more information on importing modules, see the documentation for the `import` statement in the Python Standard Library.
This module provides a very handy and useful mechanism for custom import hooks. Compared to the older `ihooks` module, `imputil` takes a dramatically simpler and more straight-forward approach to custom import functions.

```python
class imputil.ImportManager(fs_imp):
    Manage the import process.

    install([namespace])
    Install this ImportManager into the specified namespace.

    uninstall()
    Restore the previous import mechanism.

    add_suffix(suffix, importFunc)
    Undocumented.
```

```python
class imputil.Importer
    Base class for replacing standard import functions.

    import_top(name)
    Import a top-level module.

    get_code(parent, modname, fqname)
    Find and retrieve the code for the given module.

    parent specifies a parent module to define a context for
```
importing. It may be None, indicating no particular context for the search.

`modname` specifies a single module (not dotted) within the parent.

`fqname` specifies the fully-qualified module name. This is a (potentially) dotted name from the “root” of the module namespace down to the modname.

If there is no parent, then modname==fqname.

This method should return None, or a 3-tuple.

- If the module was not found, then None should be returned.
- The first item of the 2- or 3-tuple should be the integer 0 or 1, specifying whether the module that was found is a package or not.
- The second item is the code object for the module (it will be executed within the new module's namespace). This item can also be a fully-loaded module object (e.g. loaded from a shared lib).
- The third item is a dictionary of name/value pairs that will be inserted into new module before the code object is executed. This is provided in case the module's code expects certain values (such as where the module was found). When the second item is a module object, then these names/values will be inserted after the module has been loaded/initialized.

class imputil.BuiltinImporter

Emulate the import mechanism for built-in and frozen modules. This is a sub-class of the Importer class.
get_code(parent, modname, fqname)
  Undocumented.

imputil.py_suffix_importer(filename, finfo, fqname)
  Undocumented.

class imputil.DynLoadSuffixImporter([desc])
  Undocumented.

import_file(filename, finfo, fqname)
  Undocumented.
31.3.1. Examples

This is a re-implementation of hierarchical module import.

This code is intended to be read, not executed. However, it does work – all you need to do to enable it is “import knee”.

(The name is a pun on the clunkier predecessor of this module, “ni”.)

```python
import sys, imp, __builtin__

# Replacement for __import__()

def import_hook(name, globals=None, locals=None, fromlist=None):
    parent = determine_parent(globals)
    q, tail = find_head_package(parent, name)
    m = load_tail(q, tail)
    if not fromlist:
        return q
    if hasattr(m, "__path__"):
        ensure_fromlist(m, fromlist)
    return m

def determine_parent(globals):
    if not globals or not globals.has_key("__name__"):
        return None
    pname = globals['__name__']
    if globals.has_key("__path__"):
        parent = sys.modules[pname]
        assert globals is parent.__dict__
        return parent
    if '.' in pname:
        i = pname.rfind('.')
        pname = pname[:i]
        parent = sys.modules[pname]
        assert parent.__name__ == pname
        return parent
    return None

def find_head_package(parent, name):
    if '.' in name:
        i = name.find('.
        head = name[:i]
```

tail = name[i+1:]
else:
    head = name
    tail = ""
if parent:
    qname = "%s.%s" % (parent.__name__, head)
else:
    qname = head
q = import_module(head, qname, parent)
if q: return q, tail
if parent:
    qname = head
    parent = None
    q = import_module(head, qname, parent)
    if q: return q, tail
raise ImportError("No module named " + qname)

def load_tail(q, tail):
    m = q
    while tail:
        i = tail.find('.
if i < 0: i = len(tail)
head, tail = tail[:i], tail[i+1:]
mname = "%s.%s" % (m.__name__, head)
m = import_module(head, mname, m)
if not m:
    raise ImportError("No module named " + mname)
return m

def ensure_fromlist(m, fromlist, recursive=0):
    for sub in fromlist:
        if sub == "*":
            if not recursive:
                try:
                    all = m.__all__
                except AttributeError:
                    pass
                else:
            ensure_fromlist(m, all, 1)
        continue
        if sub != "*" and not hasattr(m, sub):
            subname = "%s.%s" % (m.__name__, sub)
            submod = import_module(sub, subname, m)
            if not submod:
                raise ImportError("No module named " + subname)

def import_module(partname, fqname, parent):
```python
try:
    return sys.modules[fqname]
except KeyError:
    pass
try:
    fp, pathname, stuff = imp.find_module(partname, parent and parent)
except ImportError:
    return None
try:
    m = imp.load_module(fqname, fp, pathname, stuff)
finally:
    if fp: fp.close()
    if parent:
        setattr(parent, partname, m)
return m
```

# Replacement for reload()

def reload_hook(module):
    name = module.__name__
    if '.' not in name:
        return import_module(name, name, None)
    i = name.rfind('.
    pname = name[:i]
    parent = sys.modules[pname]
    return import_module(name[i+1:], name, parent)

# Save the original hooks
original_import = __builtins__.import_
original_reload = __builtins__.reload

# Now install our hooks
__builtins__.import_ = import_hook
__builtins__.reload = reload_hook
```

Also see the importers module (which can be found in Demo/imputil/ in the Python source distribution) for additional examples.
31.4. `zipimport` — Import modules from Zip archives

*New in version 2.3.*

This module adds the ability to import Python modules (*.py, *.py[co]) and packages from ZIP-format archives. It is usually not needed to use the `zipimport` module explicitly; it is automatically used by the built-in `import` mechanism for `sys.path` items that are paths to ZIP archives.

Typically, `sys.path` is a list of directory names as strings. This module also allows an item of `sys.path` to be a string naming a ZIP file archive. The ZIP archive can contain a subdirectory structure to support package imports, and a path within the archive can be specified to only import from a subdirectory. For example, the path `example.zip/lib/` would only import from the `lib/` subdirectory within the archive.

Any files may be present in the ZIP archive, but only files `.py` and `.py[co]` are available for import. ZIP import of dynamic modules (`.pyd`, `.so`) is disallowed. Note that if an archive only contains `.py` files, Python will not attempt to modify the archive by adding the corresponding `.pyc` or `.pyo` file, meaning that if a ZIP archive doesn’t contain `.pyc` files, importing may be rather slow.

Using the built-in `reload()` function will fail if called on a module loaded from a ZIP archive; it is unlikely that `reload()` would be needed, since this would imply that the ZIP has been altered during runtime.

ZIP archives with an archive comment are currently not supported.
See also:

**PKZIP Application Note**
Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

**PEP 273 - Import Modules from Zip Archives**
Written by James C. Ahlstrom, who also provided an implementation. Python 2.3 follows the specification in PEP 273, but uses an implementation written by Just van Rossum that uses the import hooks described in PEP 302.

**PEP 302 - New Import Hooks**
The PEP to add the import hooks that help this module work.

This module defines an exception:

```python
exception zipimport.ZipImportError
    Exception raised by zipimporter objects. It's a subclass of ImportError, so it can be caught as ImportError, too.
```

### 31.4.1. zipimporter Objects

`zipimporter` is the class for importing ZIP files.

```python
class zipimport.zipimporter(archivepath)
```
Create a new zipimporter instance. `archivepath` must be a path to a ZIP file, or to a specific path within a ZIP file. For example, an `archivepath` of `foo/bar.zip/lib` will look for modules in the `lib` directory inside the ZIP file `foo/bar.zip` (provided that it exists).

`ZipImportError` is raised if `archivepath` doesn’t point to a valid ZIP archive.

```python
find_module(fullname[, path])
```
Search for a module specified by `fullname`. `fullname` must be the fully qualified (dotted) module name. It returns the zipimporter instance itself if the module was found, or `None` if it wasn’t. The optional `path` argument is ignored—it’s there for compatibility with the importer protocol.

```python
get_code(fullname)
```
Return the code object for the specified module. Raise `ZipImportError` if the module couldn’t be found.

```python
get_data(pathname)
```
Return the data associated with `pathname`. Raise `IOError` if the file wasn’t found.

```python
get_filename(fullname)
```
Return the value `__file__` would be set to if the specified module was imported. Raise `ZipImportError` if the module couldn’t be found.
New in version 2.7.

get_source(fullname)
Return the source code for the specified module. Raise ZipImportError if the module couldn't be found, return None if the archive does contain the module, but has no source for it.

is_package(fullname)
Return True if the module specified by fullname is a package. Raise ZipImportError if the module couldn’t be found.

load_module(fullname)
Load the module specified by fullname. fullname must be the fully qualified (dotted) module name. It returns the imported module, or raises ZipImportError if it wasn’t found.

archive
The file name of the importer’s associated ZIP file, without a possible subpath.

prefix
The subpath within the ZIP file where modules are searched. This is the empty string for zipimporter objects which point to the root of the ZIP file.

The archive and prefix attributes, when combined with a slash, equal the original archivepath argument given to the zipimporter constructor.
31.4.2. Examples

Here is an example that imports a module from a ZIP archive - note that the `zipimport` module is not explicitly used.

```bash
$ unzip -l example.zip
Archive: example.zip
  Length     Date   Time     Name
   --------  ----      ----      ----
       8467 11-26-02 22:30  jwzthreading.py
   --------      -------
       8467
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, 'example.zip')  # Add .zip file to front
>>> import jwzthreading
>>> jwzthreading.__file__
'example.zip/jwzthreading.py'
```
New in version 2.3.

Source code: Lib/pkgutil.py

This module provides utilities for the import system, in particular package support.

pkgutil.\texttt{extend\_path}(\textit{path}, \textit{name})

Extend the search path for the modules which comprise a package. Intended use is to place the following code in a package's \texttt{\_init\_}.py:

\begin{verbatim}
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
\end{verbatim}

This will add to the package's \texttt{\_path\_} all subdirectories of directories on \texttt{sys.path} named after the package. This is useful if one wants to distribute different parts of a single logical package as multiple directories.

It also looks for \texttt{*\_pkg} files beginning where \* matches the \textit{name} argument. This feature is similar to \texttt{*\_pth} files (see the site module for more information), except that it doesn't special-case lines starting with \texttt{import}. A \texttt{*\_pkg} file is trusted at face value: apart from checking for duplicates, all entries found in a \texttt{*\_pkg} file are added to the path, regardless of whether they exist on the filesystem. (This is a feature.)

If the input path is not a list (as is the case for frozen packages) it
is returned unchanged. The input path is not modified; an extended copy is returned. Items are only appended to the copy at the end.

It is assumed that `sys.path` is a sequence. Items of `sys.path` that are not (Unicode or 8-bit) strings referring to existing directories are ignored. Unicode items on `sys.path` that cause errors when used as filenames may cause this function to raise an exception (in line with `os.path.isdir()` behavior).

```python
class pkgutil.ImpImporter(dirname=None)

PEP 302 Importer that wraps Python’s “classic” import algorithm.

If `dirname` is a string, a PEP 302 importer is created that searches that directory. If `dirname` is `None`, a PEP 302 importer is created that searches the current `sys.path`, plus any modules that are frozen or built-in.

Note that `ImpImporter` does not currently support being used by placement on `sys.meta_path`.
```

```python
class pkgutil.ImpLoader(fullname, file, filename, etc)

PEP 302 Loader that wraps Python’s “classic” import algorithm.
```

```python
pkgutil.find_loader(fullname)

Find a PEP 302 “loader” object for `fullname`.

If `fullname` contains dots, path must be the containing package’s `__path__`. Returns `None` if the module cannot be found or imported. This function uses `iter_importers()`, and is thus subject to the same limitations regarding platform-specific special import locations such as the Windows registry.
```

```python
pkgutil.get_importer(path_item)

Retrieve a PEP 302 importer for the given `path_item`.
```
The returned importer is cached in `sys.path_importer_cache` if it was newly created by a path hook.

If there is no importer, a wrapper around the basic import machinery is returned. This wrapper is never inserted into the importer cache (None is inserted instead).

The cache (or part of it) can be cleared manually if a rescan of `sys.path_hooks` is necessary.

`pkgutil.get_loader(module_or_name)`

Get a PEP 302 “loader” object for `module_or_name`.

If the module or package is accessible via the normal import mechanism, a wrapper around the relevant part of that machinery is returned. Returns None if the module cannot be found or imported. If the named module is not already imported, its containing package (if any) is imported, in order to establish the package __path__.

This function uses `iter_importers()`, and is thus subject to the same limitations regarding platform-specific special import locations such as the Windows registry.

`pkgutil.iter_importers(fullname=``)`

Yield PEP 302 importers for the given module name.

If fullname contains a ‘.’, the importers will be for the package containing fullname, otherwise they will be importers for `sys.meta_path`, `sys.path`, and Python’s “classic” import machinery, in that order. If the named module is in a package, that package is imported as a side effect of invoking this function.

Non-PEP 302 mechanisms (e.g. the Windows registry) used by the standard import machinery to find files in alternative locations
are partially supported, but are searched after \texttt{sys.path}. Normally, these locations are searched before \texttt{sys.path}, preventing \texttt{sys.path} entries from shadowing them.

For this to cause a visible difference in behaviour, there must be a module or package name that is accessible via both \texttt{sys.path} and one of the non-\texttt{PEP 302} file system mechanisms. In this case, the emulation will find the former version, while the builtin import mechanism will find the latter.

Items of the following types can be affected by this discrepancy: \texttt{imp.C\_EXTENSION}, \texttt{imp.PY\_SOURCE}, \texttt{imp.PY\_COMPILED}, \texttt{imp.PKG\_DIRECTORY}.

\texttt{pkgutil.iter\_modules(path=None, prefix='')}

Yields \texttt{(module\_loader, name, ispkg)} for all submodules on \texttt{path}, or, if \texttt{path} is \texttt{None}, all top-level modules on \texttt{sys.path}.

\texttt{path} should be either \texttt{None} or a list of paths to look for modules in.

\texttt{prefix} is a string to output on the front of every module name on output.

\texttt{pkgutil.walk\_packages(path=None, prefix='', onerror=None)}

Yields \texttt{(module\_loader, name, ispkg)} for all modules recursively on \texttt{path}, or, if \texttt{path} is \texttt{None}, all accessible modules.

\texttt{path} should be either \texttt{None} or a list of paths to look for modules in.

\texttt{prefix} is a string to output on the front of every module name on output.

Note that this function must import all \texttt{packages} (\textit{not} all modules!)
on the given *path*, in order to access the *__path__* attribute to find submodules.

*onerror* is a function which gets called with one argument (the name of the package which was being imported) if any exception occurs while trying to import a package. If no *onerror* function is supplied, *ImportError* s are caught and ignored, while all other exceptions are propagated, terminating the search.

Examples:

```python
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ + '.')
```

**pkgutil.get_data(*package*, *resource*)**

Get a resource from a package.

This is a wrapper for the PEP 302 loader *get_data()* API. The *package* argument should be the name of a package, in standard module format (*foo.bar*). The *resource* argument should be in the form of a relative filename, using / as the path separator. The parent directory name .. is not allowed, and nor is a rooted name (starting with a /).

The function returns a binary string that is the contents of the specified resource.

For packages located in the filesystem, which have already been imported, this is the rough equivalent of:

```python
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()
```
If the package cannot be located or loaded, or it uses a **PEP 302** loader which does not support `get_data()`, then `None` is returned.

*New in version 2.6.*
31.6. modulefinder — Find modules used by a script

New in version 2.3.

Source code: Lib/modulefinder.py

This module provides a ModuleFinder class that can be used to determine the set of modules imported by a script. modulefinder.py can also be run as a script, giving the filename of a Python script as its argument, after which a report of the imported modules will be printed.

modulefinder. AddPackagePath(pkg_name, path)
Record that the package named pkg_name can be found in the specified path.

modulefinder. ReplacePackage(oldname, newname)
Allows specifying that the module named oldname is in fact the package named newname. The most common usage would be to handle how the _xmlplus package replaces the xml package.

class modulefinder. ModuleFinder([path=None, debug=0,
excludes=[], replace_paths=[]])
This class provides run_script() and report() methods to determine the set of modules imported by a script. path can be a list of directories to search for modules; if not specified, sys.path is used. debug sets the debugging level; higher values make the class print debugging messages about what it’s doing. excludes is a list of module names to exclude from the analysis. replace_paths is a list of (oldpath, newpath) tuples that will be
replaced in module paths.

**report()**
Print a report to standard output that lists the modules imported by the script and their paths, as well as modules that are missing or seem to be missing.

**run_script**(pathname)
Analyze the contents of the *pathname* file, which must contain Python code.

**modules**
A dictionary mapping module names to modules. See Example usage of ModuleFinder.
31.6.1. Example usage of ModuleFinder

The script that is going to get analyzed later on (bacon.py):

```python
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

The script that will output the report of bacon.py:

```python
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print 'Loaded modules:'
for name, mod in finder.modules.iteritems():
    print '%s: %s, %s, %s' % (name, mod.globalnames.keys()[:3])

print '-'*50
print 'Modules not imported:'
print '\n'.join(finder.badmodules.iterkeys())
```

Sample output (may vary depending on the architecture):

```
Loaded modules:
  _types:
  copy_reg: __inverted_registry, __slotnames, __all__
  sre_compile: isstring, _sre, _optimize_unicode
  _sre:
  sre_constants: REPEAT_ONE, makedict, AT_END_LINE
  sys:
```
re: __module__,finditer,_expand
itertools:
__main__: re,itertools,baconhameggs
sre_parse: __getslice__,__PATTERNENDERS,SRE_FLAG_UNICODE
array:
types: __module__,IntType,TypeType
---------------------------------------------------
Modules not imported:
guido.python.ham
baconhameggs
31.7. runpy — Locating and executing Python modules

New in version 2.5.

Source code: Lib/runpy.py

The `runpy` module is used to locate and run Python modules without importing them first. Its main use is to implement the `-m` command line switch that allows scripts to be located using the Python module namespace rather than the filesystem.

The `runpy` module provides two functions:

```python
runpy.run_module(mod_name, init_globals=None, run_name=None, alter_sys=False)
```

Execute the code of the specified module and return the resulting module globals dictionary. The module's code is first located using the standard import mechanism (refer to PEP 302 for details) and then executed in a fresh module namespace.

If the supplied module name refers to a package rather than a normal module, then that package is imported and the `__main__` submodule within that package is then executed and the resulting module globals dictionary returned.

The optional dictionary argument `init_globals` may be used to pre-populate the module's globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by `run_module()`.
The special global variables \_\_name\_\_, \_\_file\_\_, \_\_loader\_\_ and \_\_package\_\_ are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

\_\_name\_\_ is set to run\_name if this optional argument is not None, to mod\_name + \'.\_\_main\_\_' if the named module is a package and to the mod\_name argument otherwise.

\_\_file\_\_ is set to the name provided by the module loader. If the loader does not make filename information available, this variable is set to None.

\_\_loader\_\_ is set to the PEP 302 module loader used to retrieve the code for the module (This loader may be a wrapper around the standard import mechanism).

\_\_package\_\_ is set to mod\_name if the named module is a package and to mod\_name.rpartition('\.')[0] otherwise.

If the argument alter\_sys is supplied and evaluates to True, then sys.argv[0] is updated with the value of \_\_file\_\_ and sys.modules[\_\_name\_\_] is updated with a temporary module object for the module being executed. Both sys.argv[0] and sys.modules[\_\_name\_\_] are restored to their original values before the function returns.

Note that this manipulation of sys is not thread-safe. Other threads may see the partially initialised module, as well as the altered list of arguments. It is recommended that the sys module be left alone when invoking this function from threaded code.
See also: The -m option offering equivalent functionality from the command line.

Changed in version 2.7: Added ability to execute packages by looking for a __main__ submodule.

runpy.run_path(file_path, init_globals=None, run_name=None)

Execute the code at the named filesystem location and return the resulting module globals dictionary. As with a script name supplied to the CPython command line, the supplied path may refer to a Python source file, a compiled bytecode file or a valid sys.path entry containing a __main__ module (e.g. a zipfile containing a top-level __main__.py file).

For a simple script, the specified code is simply executed in a fresh module namespace. For a valid sys.path entry (typically a zipfile or directory), the entry is first added to the beginning of sys.path. The function then looks for and executes a __main__ module using the updated path. Note that there is no special protection against invoking an existing __main__ entry located elsewhere on sys.path if there is no such module at the specified location.

The optional dictionary argument init_globals may be used to pre-populate the module's globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by run_path().

The special global variables __name__, __file__, __loader__ and __package__ are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).
__name__ is set to run_name if this optional argument is not None and to '<run_path>' otherwise.

__file__ is set to the name provided by the module loader. If the loader does not make filename information available, this variable is set to None. For a simple script, this will be set to file_path.

__loader__ is set to the PEP 302 module loader used to retrieve the code for the module (This loader may be a wrapper around the standard import mechanism). For a simple script, this will be set to None.

__package__ is set to __name__.rpartition('.')[0].

A number of alterations are also made to the sys module. Firstly, sys.path may be altered as described above. sys.argv[0] is updated with the value of file_path and sys.modules[__name__] is updated with a temporary module object for the module being executed. All modifications to items in sys are reverted before the function returns.

Note that, unlike run_module(), the alterations made to sys are not optional in this function as these adjustments are essential to allowing the execution of sys.path entries. As the thread-safety limitations still apply, use of this function in threaded code should be either serialised with the import lock or delegated to a separate process.

See also: Interface options for equivalent functionality on the command line (python path/to/script).

New in version 2.7.
PEP 338 – Executing modules as scripts
  PEP written and implemented by Nick Coghlan.

PEP 366 – Main module explicit relative imports
  PEP written and implemented by Nick Coghlan.

Command line and environment - CPython command line details
32. Python Language Services

Python provides a number of modules to assist in working with the Python language. These modules support tokenizing, parsing, syntax analysis, bytecode disassembly, and various other facilities.

These modules include:

- **32.1. parser** — Access Python parse trees
  - 32.1.1. Creating ST Objects
  - 32.1.2. Converting ST Objects
  - 32.1.3. Queries on ST Objects
  - 32.1.4. Exceptions and Error Handling
  - 32.1.5. ST Objects
  - 32.1.6. Example: Emulation of `compile()`
- **32.2. ast** — Abstract Syntax Trees
  - 32.2.1. Node classes
  - 32.2.2. Abstract Grammar
  - 32.2.3. `ast` Helpers
- **32.3. symtable** — Access to the compiler’s symbol tables
  - 32.3.1. Generating Symbol Tables
  - 32.3.2. Examining Symbol Tables
- **32.4. symbol** — Constants used with Python parse trees
- **32.5. token** — Constants used with Python parse trees
- **32.6. keyword** — Testing for Python keywords
- **32.7. tokenize** — Tokenizer for Python source
- **32.8. tabnanny** — Detection of ambiguous indentation
- **32.9. pyc1br** — Python class browser support
  - 32.9.1. Class Objects
  - 32.9.2. Function Objects
- **32.10. py_compile** — Compile Python source files
- **32.11. compileall** — Byte-compile Python libraries
- 32.11.1. Command-line use
- 32.11.2. Public functions
- 32.12. dis — Disassembler for Python bytecode
  - 32.12.1. Python Bytecode Instructions
- 32.13. pickletools — Tools for pickle developers
32.1. parser — Access Python parse trees

The `parser` module provides an interface to Python’s internal parser and byte-code compiler. The primary purpose for this interface is to allow Python code to edit the parse tree of a Python expression and create executable code from this. This is better than trying to parse and modify an arbitrary Python code fragment as a string because parsing is performed in a manner identical to the code forming the application. It is also faster.

**Note:** From Python 2.5 onward, it’s much more convenient to cut in at the Abstract Syntax Tree (AST) generation and compilation stage, using the `ast` module.

The `parser` module exports the names documented here also with “st” replaced by “ast”; this is a legacy from the time when there was no other AST and has nothing to do with the AST found in Python 2.5. This is also the reason for the functions’ keyword arguments being called `ast`, not `st`. The “ast” functions have been removed in Python 3.

There are a few things to note about this module which are important to making use of the data structures created. This is not a tutorial on editing the parse trees for Python code, but some examples of using the `parser` module are presented.

Most importantly, a good understanding of the Python grammar processed by the internal parser is required. For full information on the language syntax, refer to *The Python Language Reference*. The parser itself is created from a grammar specification defined in the file `Grammar/Grammar` in the standard Python distribution. The parse
trees stored in the ST objects created by this module are the actual output from the internal parser when created by the `expr()` or `suite()` functions, described below. The ST objects created by `sequence2st()` faithfully simulate those structures. Be aware that the values of the sequences which are considered “correct” will vary from one version of Python to another as the formal grammar for the language is revised. However, transporting code from one Python version to another as source text will always allow correct parse trees to be created in the target version, with the only restriction being that migrating to an older version of the interpreter will not support more recent language constructs. The parse trees are not typically compatible from one version to another, whereas source code has always been forward-compatible.

Each element of the sequences returned by `st2list()` or `st2tuple()` has a simple form. Sequences representing non-terminal elements in the grammar always have a length greater than one. The first element is an integer which identifies a production in the grammar. These integers are given symbolic names in the C header file `Include/graminit.h` and the Python module `symbol`. Each additional element of the sequence represents a component of the production as recognized in the input string: these are always sequences which have the same form as the parent. An important aspect of this structure which should be noted is that keywords used to identify the parent node type, such as the keyword `if` in an `if_stmt`, are included in the node tree without any special treatment. For example, the `if` keyword is represented by the tuple `(1, 'if')`, where `1` is the numeric value associated with all `NAME` tokens, including variable and function names defined by the user. In an alternate form returned when line number information is requested, the same token might be represented as `(1, 'if', 12)`, where the `12` represents the line number at which the terminal symbol was found.
Terminal elements are represented in much the same way, but without any child elements and the addition of the source text which was identified. The example of the `if` keyword above is representative. The various types of terminal symbols are defined in the C header file `Include/token.h` and the Python module `token`.

The ST objects are not required to support the functionality of this module, but are provided for three purposes: to allow an application to amortize the cost of processing complex parse trees, to provide a parse tree representation which conserves memory space when compared to the Python list or tuple representation, and to ease the creation of additional modules in C which manipulate parse trees. A simple “wrapper” class may be created in Python to hide the use of ST objects.

The `parser` module defines functions for a few distinct purposes. The most important purposes are to create ST objects and to convert ST objects to other representations such as parse trees and compiled code objects, but there are also functions which serve to query the type of parse tree represented by an ST object.

See also:

**Module symbol**
Useful constants representing internal nodes of the parse tree.

**Module token**
Useful constants representing leaf nodes of the parse tree and functions for testing node values.
32.1.1. Creating ST Objects

ST objects may be created from source code or from a parse tree. When creating an ST object from source, different functions are used to create the 'eval' and 'exec' forms.

parser.expr(source)

The expr() function parses the parameter source as if it were an input to compile(source, 'file.py', 'eval'). If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

parser.suite(source)

The suite() function parses the parameter source as if it were an input to compile(source, 'file.py', 'exec'). If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

parser.sequence2st(sequence)

This function accepts a parse tree represented as a sequence and builds an internal representation if possible. If it can validate that the tree conforms to the Python grammar and all nodes are valid node types in the host version of Python, an ST object is created from the internal representation and returned to the called. If there is a problem creating the internal representation, or if the tree cannot be validated, a ParserError exception is raised. An ST object created this way should not be assumed to compile correctly; normal exceptions raised by compilation may still be initiated when the ST object is passed to compilest(). This may indicate problems not related to syntax (such as a MemoryError exception), but may also be due to constructs such as the result of parsing del f(0), which escapes the Python
parser but is checked by the bytecode compiler.

Sequences representing terminal tokens may be represented as either two-element lists of the form \((1, \text{'name'})\) or as three-element lists of the form \((1, \text{'name'}, 56)\). If the third element is present, it is assumed to be a valid line number. The line number may be specified for any subset of the terminal symbols in the input tree.

\texttt{parser.\texttt{tuple2st}(sequence)}

This is the same function as \texttt{sequence2st()}. This entry point is maintained for backward compatibility.
32.1.2. Converting ST Objects

ST objects, regardless of the input used to create them, may be converted to parse trees represented as list- or tuple- trees, or may be compiled into executable code objects. Parse trees may be extracted with or without line numbering information.

```python
parser.st2list(ast[, line_info])
```

This function accepts an ST object from the caller in `ast` and returns a Python list representing the equivalent parse tree. The resulting list representation can be used for inspection or the creation of a new parse tree in list form. This function does not fail so long as memory is available to build the list representation. If the parse tree will only be used for inspection, `st2tuple()` should be used instead to reduce memory consumption and fragmentation. When the list representation is required, this function is significantly faster than retrieving a tuple representation and converting that to nested lists.

If `line_info` is true, line number information will be included for all terminal tokens as a third element of the list representing the token. Note that the line number provided specifies the line on which the token ends. This information is omitted if the flag is false or omitted.

```python
parser.st2tuple(ast[, line_info])
```

This function accepts an ST object from the caller in `ast` and returns a Python tuple representing the equivalent parse tree. Other than returning a tuple instead of a list, this function is identical to `st2list()`.

If `line_info` is true, line number information will be included for all terminal tokens as a third element of the list representing the
token. This information is omitted if the flag is false or omitted.

```
parser.compile(ast, filename='<syntax-tree>')</n```

The Python byte compiler can be invoked on an ST object to produce code objects which can be used as part of an `exec` statement or a call to the built-in `eval()` function. This function provides the interface to the compiler, passing the internal parse tree from `ast` to the parser, using the source file name specified by the `filename` parameter. The default value supplied for `filename` indicates that the source was an ST object.

Compiling an ST object may result in exceptions related to compilation; an example would be a `SyntaxError` caused by the parse tree for `del f(θ)`: this statement is considered legal within the formal grammar for Python but is not a legal language construct. The `SyntaxError` raised for this condition is actually generated by the Python byte-compiler normally, which is why it can be raised at this point by the `parser` module. Most causes of compilation failure can be diagnosed programmatically by inspection of the parse tree.
32.1.3. Queries on ST Objects

Two functions are provided which allow an application to determine if an ST was created as an expression or a suite. Neither of these functions can be used to determine if an ST was created from source code via `expr()` or `suite()` or from a parse tree via `sequence2st()`.

**parser**. `isexpr(ast)`

When `ast` represents an 'eval' form, this function returns true, otherwise it returns false. This is useful, since code objects normally cannot be queried for this information using existing built-in functions. Note that the code objects created by `compilest()` cannot be queried like this either, and are identical to those created by the built-in `compile()` function.

**parser**. `issuite(ast)`

This function mirrors `isexpr()` in that it reports whether an ST object represents an 'exec' form, commonly known as a “suite.” It is not safe to assume that this function is equivalent to `not isexpr(ast)`, as additional syntactic fragments may be supported in the future.
32.1.4. Exceptions and Error Handling

The parser module defines a single exception, but may also pass other built-in exceptions from other portions of the Python runtime environment. See each function for information about the exceptions it can raise.

\textit{exception} \texttt{parser.ParserError}

Exception raised when a failure occurs within the parser module. This is generally produced for validation failures rather than the built-in \texttt{SyntaxError} raised during normal parsing. The exception argument is either a string describing the reason of the failure or a tuple containing a sequence causing the failure from a parse tree passed to \texttt{sequence2st()} and an explanatory string. Calls to \texttt{sequence2st()} need to be able to handle either type of exception, while calls to other functions in the module will only need to be aware of the simple string values.

Note that the functions \texttt{compilest()}, \texttt{expr()}, and \texttt{suite()} may raise exceptions which are normally raised by the parsing and compilation process. These include the built in exceptions \texttt{MemoryError}, \texttt{OverflowError}, \texttt{SyntaxError}, and \texttt{SystemError}. In these cases, these exceptions carry all the meaning normally associated with them. Refer to the descriptions of each function for detailed information.
32.1.5. ST Objects

Ordered and equality comparisons are supported between ST objects. Pickling of ST objects (using the `pickle` module) is also supported.

**parser.STType**

The type of the objects returned by `expr()`, `suite()` and `sequence2st()`.

ST objects have the following methods:

**ST.compile([filename])**

Same as `compilest(st, filename)`.  

**ST.isexpr()**

Same as `isexpr(st)`.  

**ST.issuite()**

Same as `issuite(st)`.  

**ST.tolist([line_info])**

Same as `st2list(st, line_info)`.  

**ST.totuple([line_info])**

Same as `st2tuple(st, line_info)`.  

32.1.6. Example: Emulation of `compile()`

While many useful operations may take place between parsing and bytecode generation, the simplest operation is to do nothing. For this purpose, using the `parser` module to produce an intermediate data structure is equivalent to the code

```python
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

The equivalent operation using the `parser` module is somewhat longer, and allows the intermediate internal parse tree to be retained as an ST object:

```python
>>> import parser
>>> st = parser.expr('a + 5')
>>> code = st.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

An application which needs both ST and code objects can package this code into readily available functions:

```python
import parser

def load_suite(source_string):
    st = parser.suite(source_string)
    return st, st.compile()

def load_expression(source_string):
    st = parser.expr(source_string)
    return st, st.compile()
```
32.2. ast — Abstract Syntax Trees

New in version 2.5: The low-level _ast module containing only the node classes.

New in version 2.6: The high-level ast module containing all helpers.

**Source code:** Lib/ast.py

The ast module helps Python applications to process trees of the Python abstract syntax grammar. The abstract syntax itself might change with each Python release; this module helps to find out programatically what the current grammar looks like.

An abstract syntax tree can be generated by passing ast.PyCF_ONLY_AST as a flag to the compile() built-in function, or using the parse() helper provided in this module. The result will be a tree of objects whose classes all inherit from ast.AST. An abstract syntax tree can be compiled into a Python code object using the built-in compile() function.
32.2.1. Node classes

```python
class ast.AST
    This is the base of all AST node classes. The actual node classes are derived from the Parser/Python.asdl file, which is reproduced below. They are defined in the _ast C module and re-exported in ast.
```

There is one class defined for each left-hand side symbol in the abstract grammar (for example, `ast.stmt` or `ast.expr`). In addition, there is one class defined for each constructor on the right-hand side; these classes inherit from the classes for the left-hand side trees. For example, `ast.BinOp` inherits from `ast.expr`. For production rules with alternatives (aka “sums”), the left-hand side class is abstract: only instances of specific constructor nodes are ever created.

_`fields`

Each concrete class has an attribute `_fields` which gives the names of all child nodes.

Each instance of a concrete class has one attribute for each child node, of the type as defined in the grammar. For example, `ast.BinOp` instances have an attribute `left` of type `ast.expr`.

If these attributes are marked as optional in the grammar (using a question mark), the value might be `None`. If the attributes can have zero-or-more values (marked with an asterisk), the values are represented as Python lists. All possible attributes must be present and have valid values when compiling an AST with `compile()`.
Instances of `ast.expr` and `ast.stmt` subclasses have `lineno` and `col_offset` attributes. The `lineno` is the line number of source text (1-indexed so the first line is line 1) and the `col_offset` is the UTF-8 byte offset of the first token that generated the node. The UTF-8 offset is recorded because the parser uses UTF-8 internally.

The constructor of a class `ast.T` parses its arguments as follows:

- If there are positional arguments, there must be as many as there are items in `T._fields`; they will be assigned as attributes of these names.
- If there are keyword arguments, they will set the attributes of the same names to the given values.

For example, to create and populate an `ast.UnaryOp` node, you could use

```python
node = ast.UnaryOp()
node.op = ast.USub()
node.operand = ast.Num()
nombre.operand.n = 5
node.operand.lineno = 0
node.operand.col_offset = 0
node.lineno = 0
node.col_offset = 0
```

or the more compact

```python
node = ast.UnaryOp(ast.USub(), ast.Num(5, lineno=0, col_offset=0)
```

New in version 2.6: The constructor as explained above was added. In Python 2.5 nodes had to be created by calling the class constructor without arguments and setting the attributes
afterwards.
32.2.2. Abstract Grammar

The module defines a string constant \_version\_ which is the decimal Subversion revision number of the file shown below.

The abstract grammar is currently defined as follows:

```
module Python version "\$Revision\$
{
    mod = Module(stmt* body)
    | Interactive(stmt* body)
    | Expression(expr body)

    -- not really an actual node but useful in Jython's
    | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
        stmt* body, expr* decorator_list)
    | ClassDef(identifier name, expr* bases, stmt* body,
        expr* decorator_list)
    | Return(expr? value)
    | Delete(expr* targets)
    | Assign(expr* targets, expr value)
    | AugAssign(expr target, operator op, expr value)

    -- not sure if bool is allowed, can always use in
    | Print(expr? dest, expr* values, bool nl)

    -- use 'orelse' because else is a keyword in target
    | For(expr target, expr iter, stmt* body, stmt* orelse)
    | While(expr test, stmt* body, stmt* orelse)
    | If(expr test, stmt* body, stmt* orelse)
    | With(expr context_expr, expr? optional_vars, stmt*

    -- 'type' is a bad name
    | Raise(expr? type, expr? inst, expr? tback)
    | TryExcept(stmt* body, excepthandler* handlers,
        TryFinally(stmt* body, stmt* finalbody)
    | Assert(expr test, expr? msg)
    | Import(alias* names)
```
ImportFrom(identifier? module, alias* names, in
| Exec(expr body, expr? globals, expr? locals)
| Global(identifier* names)
| Expr(expr value)
| Pass | Break | Continue

-- XXX Jython will be different
-- col_offset is the byte offset in the utf8 string attributes (int lineno, int col_offset)

-- BoolOp() can use left & right?
expr = BoolOp(boolop op, expr* values)
| BinOp(expr left, operator op, expr right)
| UnaryOp(unaryop op, expr operand)
| Lambda(arguments args, expr body)
| IfExp(expr test, expr body, expr orelse)
| Dict(expr* keys, expr* values)
| Set(expr* elts)
| ListComp(expr elt, comprehension* generators)
| SetComp(expr elt, comprehension* generators)
| DictComp(expr key, expr value, comprehension* generators)
| GeneratorExp(expr elt, comprehension* generators)
-- the grammar constrains where yield expressions
| Yield(expr? value)
-- need sequences for compare to distinguish between
-- x < 4 < 3 and (x < 4) < 3
| Compare(expr left, cmpop* ops, expr* comparators
| Call(expr func, expr* args, keyword* keywords,
  expr? starargs, expr? kwargs)
| Repr(expr value)
| Num(object n) -- a number as a PyObject.
| Str(string s) -- need to specify raw, unicode, e
-- other literals? bools?
-- the following expression can appear in assignment
| Attribute(expr value, identifier attr, expr_context
| Subscript(expr value, slice slice, expr_context
| Name(identifier id, expr_context ctx)
| List(expr* elts, expr_context ctx)
| Tuple(expr* elts, expr_context ctx)

-- col_offset is the byte offset in the utf8 stri
attributes (int lineno, int col_offset)
expr_context = Load | Store | Del | AugLoad | AugStore

slice = Ellipsis | Slice(expr? lower, expr? upper, expr?) | ExtSlice(slice* dims) | Index(expr value)

boolop = And | Or
operator = Add | Sub | Mult | Div | Mod | Pow | LShift | RShift | BitOr | BitXor | BitAnd | FloorDiv
unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot |
comprehension = (expr target, expr iter, expr* ifs)

-- not sure what to call the first argument for raise a
excepthandler = ExceptHandler(expr? type, expr? name, stmt* body)

arguments = (expr* args, identifier? vararg, identifier? kwarg, expr* defaults)

-- keyword arguments supplied to call
keyword = (identifier arg, expr value)

-- import name with optional 'as' alias.
alias = (identifier name, identifier? asname)
32.2.3. ast Helpers

*New in version 2.6.*

Apart from the node classes, ast module defines these utility functions and classes for traversing abstract syntax trees:

```python
ast.parse(source, filename='<unknown>', mode='exec')
```

Parse the source into an AST node. Equivalent to `compile(source, filename, mode, ast.PyCF_ONLY_AST)`.

```python
ast.literal_eval(node_or_string)
```

Safely evaluate an expression node or a Unicode or *Latin-1* encoded string containing a Python literal or container display. The string or node provided may only consist of the following Python literal structures: strings, numbers, tuples, lists, dicts, booleans, and `None`.

This can be used for safely evaluating strings containing Python values from untrusted sources without the need to parse the values oneself. It is not capable of evaluating arbitrarily complex expressions, for example involving operators or indexing.

```python
ast.get_docstring(node, clean=True)
```

Return the docstring of the given `node` (which must be a `FunctionDef`, `ClassDef` or `Module` node), or `None` if it has no docstring. If `clean` is true, clean up the docstring’s indentation with `inspect.cleandoc()`.

```python
ast.fix_missing_locations(node)
```

When you compile a node tree with `compile()`, the compiler expects `lineno` and `col_offset` attributes for every node that supports them. This is rather tedious to fill in for generated
nodes, so this helper adds these attributes recursively where not already set, by setting them to the values of the parent node. It works recursively starting at node.

```
ast.increment_lineno(node, n=1)
```
Increment the line number of each node in the tree starting at node by n. This is useful to “move code” to a different location in a file.

```
ast.copy_location(new_node, old_node)
```
Copy source location (lineno and col_offset) from old_node to new_node if possible, and return new_node.

```
ast.iter_fields(node)
```
Yield a tuple of (fieldname, value) for each field in node._fields that is present on node.

```
ast.iter_child_nodes(node)
```
Yield all direct child nodes of node, that is, all fields that are nodes and all items of fields that are lists of nodes.

```
ast.walk(node)
```
Recursively yield all descendant nodes in the tree starting at node (including node itself), in no specified order. This is useful if you only want to modify nodes in place and don’t care about the context.

```
class ast.NodeVisitor
```
A node visitor base class that walks the abstract syntax tree and calls a visitor function for every node found. This function may return a value which is forwarded by the visit() method.

This class is meant to be subclassed, with the subclass adding visitor methods.
visit(node)

Visit a node. The default implementation calls the method called `self.visit_classname` where `classname` is the name of the node class, or `generic_visit()` if that method doesn’t exist.

generic_visit(node)

This visitor calls `visit()` on all children of the node.

Note that child nodes of nodes that have a custom visitor method won’t be visited unless the visitor calls `generic_visit()` or visits them itself.

Don’t use the `NodeVisitor` if you want to apply changes to nodes during traversal. For this a special visitor exists (`NodeTransformer`) that allows modifications.

class ast.NodeTransformer

A `NodeVisitor` subclass that walks the abstract syntax tree and allows modification of nodes.

The `NodeTransformer` will walk the AST and use the return value of the visitor methods to replace or remove the old node. If the return value of the visitor method is `None`, the node will be removed from its location, otherwise it is replaced with the return value. The return value may be the original node in which case no replacement takes place.

Here is an example transformer that rewrites all occurrences of name lookups (`foo`) to `data['foo']`:

```python
class RewriteName(NodeTransformer):
    def visit_Name(self, node):
        return copy_location(Subscript(  
```
Keep in mind that if the node you're operating on has child nodes you must either transform the child nodes yourself or call the `generic_visit()` method for the node first.

For nodes that were part of a collection of statements (that applies to all statement nodes), the visitor may also return a list of nodes rather than just a single node.

Usually you use the transformer like this:

```python
node = YourTransformer().visit(node)
```

```
 ast.dump(node, annotate_fields=True, include_attributes=False)
```

Return a formatted dump of the tree in `node`. This is mainly useful for debugging purposes. The returned string will show the names and the values for fields. This makes the code impossible to evaluate, so if evaluation is wanted `annotate_fields` must be set to `False`. Attributes such as line numbers and column offsets are not dumped by default. If this is wanted, `include_attributes` can be set to `True`. 
32.3. symtable — Access to the compiler’s symbol tables

**Source code:** Lib/symtable.py

Symbol tables are generated by the compiler from AST just before bytecode is generated. The symbol table is responsible for calculating the scope of every identifier in the code. *symtable* provides an interface to examine these tables.
32.3.1. Generating Symbol Tables

`symtable.symtable(code, filename, compile_type)`

Return the toplevel `SymbolTable` for the Python source `code`. `filename` is the name of the file containing the code. `compile_type` is like the `mode` argument to `compile()`.
32.3.2. Examining Symbol Tables

class symtable.SymbolTable
A namespace table for a block. The constructor is not public.

get_type()
Return the type of the symbol table. Possible values are 'class', 'module', and 'function'.

get_id()
Return the table’s identifier.

get_name()
Return the table’s name. This is the name of the class if the table is for a class, the name of the function if the table is for a function, or 'top' if the table is global (get_type() returns 'module').

get_lineno()
Return the number of the first line in the block this table represents.

is_optimized()
Return True if the locals in this table can be optimized.

is_nested()
Return True if the block is a nested class or function.

has_children()
Return True if the block has nested namespaces within it. These can be obtained with get_children().

has_exec()
Return `True` if the block uses `exec`.

`has_import_star()`
Return `True` if the block uses a starred from-import.

`get_identifiers()`
Return a list of names of symbols in this table.

`lookup(name)`
Lookup `name` in the table and return a `Symbol` instance.

`get_symbols()`
Return a list of `Symbol` instances for names in the table.

`get_children()`
Return a list of the nested symbol tables.

class `symtable.Function`
A namespace for a function or method. This class inherits `SymbolTable`.

`get_parameters()`
Return a tuple containing names of parameters to this function.

`get_locals()`
Return a tuple containing names of locals in this function.

`getGlobals()`
Return a tuple containing names of globals in this function.

`get_frees()`
Return a tuple containing names of free variables in this function.
class symtable.

**Class**

A namespace of a class. This class inherits `SymbolTable`.

**get_methods()**

Return a tuple containing the names of methods declared in the class.

**class symtable. Symbol**

An entry in a `SymbolTable` corresponding to an identifier in the source. The constructor is not public.

**get_name()**

Return the symbol’s name.

**is_referenced()**

Return `True` if the symbol is used in its block.

**is_imported()**

Return `True` if the symbol is created from an import statement.

**is_parameter()**

Return `True` if the symbol is a parameter.

**is_global()**

Return `True` if the symbol is global.

**isDeclaredGlobal()**

Return `True` if the symbol is declared global with a global statement.

**is_local()**

Return `True` if the symbol is local to its block.

**is_free()**
Return `True` if the symbol is referenced in its block, but not assigned to.

### is_assigned()

Return `True` if the symbol is assigned to in its block.

### is_namespace()

Return `True` if name binding introduces new namespace.

If the name is used as the target of a function or class statement, this will be true.

For example:

```python
gt>> table = symtable.symtable("def some_func(): pass", "string"
<...>
gt>> table.lookup("some_func").is_namespace()
True
```

Note that a single name can be bound to multiple objects. If the result is `True`, the name may also be bound to other objects, like an int or list, that does not introduce a new namespace.

### get_namespaces()

Return a list of namespaces bound to this name.

### get_namespace()

Return the namespace bound to this name. If more than one namespace is bound, a `ValueError` is raised.
This module provides constants which represent the numeric values of internal nodes of the parse tree. Unlike most Python constants, these use lower-case names. Refer to the file Grammar/Grammar in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

This module also provides one additional data object:

```
symbol. sym_name
```

Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing more human-readable representation of parse trees to be generated.
32.5. token — Constants used with Python parse trees

**Source code:** Lib/token.py

This module provides constants which represent the numeric values of leaf nodes of the parse tree (terminal tokens). Refer to the file `Grammar/Grammar` in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

The module also provides a mapping from numeric codes to names and some functions. The functions mirror definitions in the Python C header files.

**token. tok_name**

Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing more human-readable representation of parse trees to be generated.

**token. ISTERMINAL(x)**

Return true for terminal token values.

**token. ISNONTERMINAL(x)**

Return true for non-terminal token values.

**token. ISEOF(x)**

Return true if x is the marker indicating the end of input.

The token constants are:

**token. ENDMARKER**

**token. NAME**
See also:

**Module** `parser`

The second example for the `parser` module shows how to use the `symbol` module.
This module allows a Python program to determine if a string is a keyword.

```python
keyword.iskeyword(s)
```

Return true if s is a Python keyword.

```python
keyword.kwlist
```

Sequence containing all the keywords defined for the interpreter. If any keywords are defined to only be active when particular `__future__` statements are in effect, these will be included as well.
32.7. `tokenize` — Tokenizer for Python source

**Source code:** `Lib/tokenize.py`

The `tokenize` module provides a lexical scanner for Python source code, implemented in Python. The scanner in this module returns comments as tokens as well, making it useful for implementing “pretty-printers,” including colorizers for on-screen displays.

To simplify token stream handling, all Operators and Delimiters tokens are returned using the generic `token.OP` token type. The exact type can be determined by checking the second field (containing the actual token string matched) of the tuple returned from `tokenize.generate_tokens()` for the character sequence that identifies a specific operator token.

The primary entry point is a generator:

```python
tokenizer.generate_tokens(readline)
```

The `generate_tokens()` generator requires one argument, `readline`, which must be a callable object which provides the same interface as the `readline()` method of built-in file objects (see section File Objects). Each call to the function should return one line of input as a string. Alternately, `readline` may be a callable object that signals completion by raising `StopIteration`.

The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple `(srow, scol)` of ints specifying the row and column where the token begins in the source; a 2-tuple `(erow, ecol)` of ints specifying the row and column where the token ends in the source; and the line on which the token was
found. The line passed (the last tuple item) is the *logical* line; continuation lines are included.

*New in version 2.2.*

An older entry point is retained for backward compatibility:

```python
tokenize(tokenize(readline[, tokeneater])
```

The `tokenize()` function accepts two parameters: one representing the input stream, and one providing an output mechanism for `tokenize()`.

The first parameter, `readline`, must be a callable object which provides the same interface as the `readline()` method of built-in file objects (see section *File Objects*). Each call to the function should return one line of input as a string. Alternately, `readline` may be a callable object that signals completion by raising `StopIteration`.

*Changed in version 2.5:* Added `StopIteration` support.

The second parameter, `tokeneater`, must also be a callable object. It is called once for each token, with five arguments, corresponding to the tuples generated by `generate_tokens()`.

All constants from the `token` module are also exported from `tokenize`, as are two additional token type values that might be passed to the `tokeneater` function by `tokenize()`:

```python
tokenize.COMMENT
    Token value used to indicate a comment.
    
tokenize.NL
    Token value used to indicate a non-terminating newline. The NEWLINE token indicates the end of a logical line of Python
code; NL tokens are generated when a logical line of code is
continued over multiple physical lines.
Another function is provided to reverse the tokenization process.
This is useful for creating tools that tokenize a script, modify the
token stream, and write back the modified script.
tokenize. untokenize (iterable)

Converts tokens back into Python source code. The iterable must
return sequences with at least two elements, the token type and
the token string. Any additional sequence elements are ignored.
The reconstructed script is returned as a single string. The result
is guaranteed to tokenize back to match the input so that the
conversion is lossless and round-trips are assured. The
guarantee applies only to the token type and token string as the
spacing between tokens (column positions) may change.
New in version 2.5.
exception tokenize. TokenError
Raised when either a docstring or expression that may be split
over several lines is not completed anywhere in the file, for
example:
"""Beginning of
docstring

or:
[1,
2,
3

Note that unclosed single-quoted strings do not cause an error to be
raised. They are tokenized as ERRORTOKEN , followed by the
tokenization of their contents.


Example of a script re-writer that transforms float literals into Decimal objects:

```python
def decistmt(s):
    """Substitute Decimals for floats in a string of statements."""

    >> from decimal import Decimal
    >> s = 'print +21.3e-5*-1234/81.7'
    >> decistmt(s)
    "print +Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7')"'

    >>> exec(s)
    -3.21716034272e-007
    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7

    ""
    result = []
    g = generate_tokens(StringIO(s).readline)  # tokenize the
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER
            result.extend([
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
            result.append((toknum, tokval))
    return untokenize(result)
```

32.8. **tabnanny — Detection of ambiguous indentation**

**Source code:** Lib/tabnanny.py

For the time being this module is intended to be called as a script. However it is possible to import it into an IDE and use the function `check()` described below.

**Note:** The API provided by this module is likely to change in future releases; such changes may not be backward compatible.

```python
def check(file_or_dir):
    if file_or_dir is a directory and not a symbolic link, then recursively descend the directory tree named by file_or_dir, checking all .py files along the way. If file_or_dir is an ordinary Python source file, it is checked for whitespace related problems. The diagnostic messages are written to standard output using the print statement.
```

**tabnanny.** `verbose`

Flag indicating whether to print verbose messages. This is incremented by the `-v` option if called as a script.

**tabnanny.** `filename_only`

Flag indicating whether to print only the filenames of files containing whitespace related problems. This is set to true by the `-q` option if called as a script.

**exception** `tabnanny.NannyNag`

Raised by `tokeneater()` if detecting an ambiguous indent. Captured and handled in `check()`.
tabnanny. **tokeneater**(type, token, start, end, line)

This function is used by `check()` as a callback parameter to the function `tokenize.tokenize()`.

**See also:**

**Module** `tokenize`

Lexical scanner for Python source code.
The `pyclbr` module can be used to determine some limited information about the classes, methods and top-level functions defined in a module. The information provided is sufficient to implement a traditional three-pane class browser. The information is extracted from the source code rather than by importing the module, so this module is safe to use with untrusted code. This restriction makes it impossible to use this module with modules not implemented in Python, including all standard and optional extension modules.

`pyclbr.readmodule(module, path=None)`

Read a module and return a dictionary mapping class names to class descriptor objects. The parameter `module` should be the name of a module as a string; it may be the name of a module within a package. The `path` parameter should be a sequence, and is used to augment the value of `sys.path`, which is used to locate module source code.

`pyclbr.readmodule_ex(module, path=None)`

Like `readmodule()`, but the returned dictionary, in addition to mapping class names to class descriptor objects, also maps top-level function names to function descriptor objects. Moreover, if the module being read is a package, the key `__path__` in the returned dictionary has as its value a list which contains the package search path.
32.9.1. Class Objects

The class objects used as values in the dictionary returned by readmodule() and readmodule_ex() provide the following data attributes:

**Class.module**

The name of the module defining the class described by the class descriptor.

**Class.name**

The name of the class.

**Class.super**

A list of class objects which describe the immediate base classes of the class being described. Classes which are named as superclasses but which are not discoverable by readmodule() are listed as a string with the class name instead of as class objects.

**Class.methods**

A dictionary mapping method names to line numbers.

**Class.file**

Name of the file containing the class statement defining the class.

**Class.lineno**

The line number of the class statement within the file named by file.
32.9.2. Function Objects

The `Function` objects used as values in the dictionary returned by `readmodule_ex()` provide the following attributes:

Function. `module`

- The name of the module defining the function described by the function descriptor.

Function. `name`

- The name of the function.

Function. `file`

- Name of the file containing the `def` statement defining the function.

Function. `lineno`

- The line number of the `def` statement within the file named by `file`.
32.10. py_compile — Compile Python source files

Source code: Lib/py_compile.py

The `py_compile` module provides a function to generate a byte-code file from a source file, and another function used when the module source file is invoked as a script.

Though not often needed, this function can be useful when installing modules for shared use, especially if some of the users may not have permission to write the byte-code cache files in the directory containing the source code.

```
exception py_compile.PyCompileError
    Exception raised when an error occurs while attempting to compile the file.
```

```
py_compile.compile(file[, cfile[, dfile[, doraise]]])
    Compile a source file to byte-code and write out the byte-code cache file. The source code is loaded from the file named `file`. The byte-code is written to `cfile`, which defaults to `file + 'c'` (`'o'` if optimization is enabled in the current interpreter). If `dfile` is specified, it is used as the name of the source file in error messages instead of `file`. If `doraise` is true, a PyCompileError is raised when an error is encountered while compiling `file`. If `doraise` is false (the default), an error string is written to `sys.stderr`, but no exception is raised.
```

```
py_compile.main([args])
    Compile several source files. The files named in `args` (or on the
```
command line, if *args* is not specified) are compiled and the resulting bytecode is cached in the normal manner. This function does not search a directory structure to locate source files; it only compiles files named explicitly. If `-` is the only parameter in *args*, the list of files is taken from standard input.

*Changed in version 2.7:* Added support for `-`.

When this module is run as a script, the `main()` is used to compile all the files named on the command line. The exit status is nonzero if one of the files could not be compiled.

*Changed in version 2.6:* Added the nonzero exit status when module is run as a script.

**See also:**

**Module** *compileall*

Utilities to compile all Python source files in a directory tree.
This module provides some utility functions to support installing Python libraries. These functions compile Python source files in a directory tree. This module can be used to create the cached bytecode files at library installation time, which makes them available for use even by users who don’t have write permission to the library directories.
32.11.1. Command-line use

This module can work as a script (using `python -m compileall`) to compile Python sources.

```markdown
**directory ...**
**file ...**
```

Positional arguments are files to compile or directories that contain source files, traversed recursively. If no argument is given, behave as if the command line was `-l <directories from sys.path>`.

- `-l`
  Do not recurse into subdirectories, only compile source code files directly contained in the named or implied directories.

- `-f`
  Force rebuild even if timestamps are up-to-date.

- `-q`
  Do not print the list of files compiled, print only error messages.

- `-d destdir`
  Directory prepended to the path to each file being compiled. This will appear in compilation time tracebacks, and is also compiled into the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

- `-x regex`
  regex is used to search the full path to each file considered for compilation, and if the regex produces a match, the file is skipped.

- `-i list`
Read the file list and add each line that it contains to the list of files and directories to compile. If list is -, read lines from stdin.

*Changed in version 2.7:* Added the `-i` option.
compileall.compile_dir(dir[, maxlevels[, ddir[, force[, rx[, quiet]]]]])

Recursively descend the directory tree named by dir, compiling all .py files along the way.

The maxlevels parameter is used to limit the depth of the recursion; it defaults to 10.

If ddir is given, it is prepended to the path to each file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If force is true, modules are re-compiled even if the timestamps are up to date.

If rx is given, its search method is called on the complete path to each file considered for compilation, and if it returns a true value, the file is skipped.

If quiet is true, nothing is printed to the standard output unless errors occur.

compileall.compile_file(fullname[, ddir[, force[, rx[, quiet]]]])

Compile the file with path fullname.

If ddir is given, it is prepended to the path to the file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file
does not exist at the time the byte-code file is executed.

If \( rx \) is given, its search method is passed the full path name to the file being compiled, and if it returns a true value, the file is not compiled and \texttt{True} is returned.

If \texttt{quiet} is true, nothing is printed to the standard output unless errors occur.

\textit{New in version 2.7.}

\texttt{compileall.compile\_path([\texttt{skip\_curdir}, \texttt{maxlevels}, \texttt{force}])}

Byte-compile all the .py files found along \texttt{sys.path}. If \texttt{skip\_curdir} is true (the default), the current directory is not included in the search. All other parameters are passed to the \texttt{compile\_dir()} function. Note that unlike the other compile functions, \texttt{maxlevels} defaults to 0.

To force a recompile of all the .py files in the \texttt{Lib/} subdirectory and all its subdirectories:

```python
import compileall
compileall.compile_dir('Lib/', force=True)
```

# Perform same compilation, excluding files in .svn directories
```
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\][.]svn'), f
```

\textbf{See also:}

\textbf{Module} \textit{py\_compile}

Byte-compile a single source file.
32.12. dis — Disassembler for Python bytecode

Source code: Lib/dis.py

The dis module supports the analysis of CPython bytecode by disassembling it. The CPython bytecode which this module takes as an input is defined in the file Include/opcode.h and used by the compiler and the interpreter.

**CPython implementation detail:** Bytecode is an implementation detail of the CPython interpreter! No guarantees are made that bytecode will not be added, removed, or changed between versions of Python. Use of this module should not be considered to work across Python VMs or Python releases.

Example: Given the function myfunc():

```python
def myfunc(alist):
    return len(alist)
```

the following command can be used to get the disassembly of myfunc():

```python
>>> dis.dis(myfunc)
  2           0 LOAD_GLOBAL              0 (len)
  3           3 LOAD_FAST                0 (alist)
  6           6 CALL_FUNCTION            1
  9           9 RETURN_VALUE
```

(The “2” is a line number).
The `dis` module defines the following functions and constants:

**dis.dis([`bytesource`])**
Disassemble the `bytesource` object. `bytesource` can denote either a module, a class, a method, a function, or a code object. For a module, it disassembles all functions. For a class, it disassembles all methods. For a single code sequence, it prints one line per bytecode instruction. If no object is provided, it disassembles the last traceback.

**dis.distb([`tb`])**
Disassemble the top-of-stack function of a traceback, using the last traceback if none was passed. The instruction causing the exception is indicated.

**dis.disassemble(`code`, `lasti`)**
Disassembles a code object, indicating the last instruction if `lasti` was provided. The output is divided in the following columns:

1. the line number, for the first instruction of each line
2. the current instruction, indicated as `-->`,
3. a labelled instruction, indicated with `>>`,
4. the address of the instruction,
5. the operation code name,
6. operation parameters, and
7. interpretation of the parameters in parentheses.
The parameter interpretation recognizes local and global variable names, constant values, branch targets, and compare operators.

**dis.disco(`code`, `lasti`)**
A synonym for `disassemble()`. It is more convenient to type, and kept for compatibility with earlier Python releases.

**dis.findlinestarts(`code`)**
This generator function uses the `co_firstlineno` and `co_lnotab` attributes of the code object `code` to find the offsets which are starts of lines in the source code. They are generated as `(offset, lineno)` pairs.

```
dis.findlabels(code)
Detect all offsets in the code object `code` which are jump targets, and return a list of these offsets.
```

```
dis.opname
Sequence of operation names, indexable using the bytecode.
```

```
dis.opmap
Dictionary mapping operation names to bytecodes.
```

```
dis.cmp_op
Sequence of all compare operation names.
```

```
dis.hasconst
Sequence of bytecodes that have a constant parameter.
```

```
dis.hasfree
Sequence of bytecodes that access a free variable.
```

```
dis.hasname
Sequence of bytecodes that access an attribute by name.
```

```
dis.hasjrel
Sequence of bytecodes that have a relative jump target.
```

```
dis.hasjabs
Sequence of bytecodes that have an absolute jump target.
```

```
dis.haslocal
Sequence of bytecodes that access a local variable.
```

```
dis.hascompare
```
Sequence of bytecodes of Boolean operations.
32.12.1. Python Bytecode Instructions

The Python compiler currently generates the following bytecode instructions.

**STOP_CODE()**
Indicates end-of-code to the compiler, not used by the interpreter.

**NOP()**
Do nothing code. Used as a placeholder by the bytecode optimizer.

**POP_TOP()**
Removes the top-of-stack (TOS) item.

**ROT_TWO()**
Swaps the two top-most stack items.

**ROT_THREE()**
Lifts second and third stack item one position up, moves top down to position three.

**ROT_FOUR()**
Lifts second, third and forth stack item one position up, moves top down to position four.

**DUP_TOP()**
Duplicates the reference on top of the stack.

Unary Operations take the top of the stack, apply the operation, and push the result back on the stack.

**UNARY_POSITIVE()**
Implements $TOS = +TOS$. 
UNARY_NEGATIVE()
    Implements \( TOS = -TOS \).

UNARY_NOT()
    Implements \( TOS = \text{not } TOS \).

UNARY_CONVERT()
    Implements \( TOS = \text{`TOS`} \).

UNARY_INVERT()
    Implements \( TOS = \text{~TOS} \).

GET_ITER()
    Implements \( TOS = \text{iter(TOS)} \).

Binary operations remove the top of the stack (TOS) and the second top-most stack item (TOS1) from the stack. They perform the operation, and put the result back on the stack.

BINARY_POWER()
    Implements \( TOS = TOS1 ** TOS \).

BINARY_MULTIPLY()
    Implements \( TOS = TOS1 * TOS \).

BINARY_DIVIDE()
    Implements \( TOS = TOS1 / TOS \) when from __future__ import division is not in effect.

BINARY_FLOOR_DIVIDE()
    Implements \( TOS = TOS1 // TOS \).

BINARY_TRUE_DIVIDE()
    Implements \( TOS = TOS1 / TOS \) when from __future__ import division is in effect.
**BINARY_MODULO()**
Implements $TOS = TOS1 \% TOS$.

**BINARY_ADD()**
Implements $TOS = TOS1 + TOS$.

**BINARY_SUBTRACT()**
Implements $TOS = TOS1 - TOS$.

**BINARY_SUBSCR()**
Implements $TOS = TOS1[TOS]$.

**BINARY_LSHIFT()**
Implements $TOS = TOS1 << TOS$.

**BINARY_RSHIFT()**
Implements $TOS = TOS1 >> TOS$.

**BINARY_AND()**
Implements $TOS = TOS1 \& TOS$.

**BINARY_XOR()**
Implements $TOS = TOS1 ^ TOS$.

**BINARY_OR()**
Implements $TOS = TOS1 | TOS$.

In-place operations are like binary operations, in that they remove TOS and TOS1, and push the result back on the stack, but the operation is done in-place when TOS1 supports it, and the resulting TOS may be (but does not have to be) the original TOS1.

**INPLACE_POWER()**
Implements in-place $TOS = TOS1 ** TOS$. 
**INPLACE_MULTIPLY()**
Implements in-place $\text{TOS} = \text{TOS1} \times \text{TOS}$.

**INPLACE_DIVIDE()**
Implements in-place $\text{TOS} = \text{TOS1} / \text{TOS}$ when `from __future__ import division` is not in effect.

**INPLACE_FLOOR_DIVIDE()**
Implements in-place $\text{TOS} = \text{TOS1} \div \text{TOS}$.

**INPLACE_TRUE_DIVIDE()**
Implements in-place $\text{TOS} = \text{TOS1} / \text{TOS}$ when `from __future__ import division` is in effect.

**INPLACE_MODULO()**
Implements in-place $\text{TOS} = \text{TOS1} \mod \text{TOS}$.

**INPLACE_ADD()**
Implements in-place $\text{TOS} = \text{TOS1} + \text{TOS}$.

**INPLACE_SUBTRACT()**
Implements in-place $\text{TOS} = \text{TOS1} - \text{TOS}$.

**INPLACE_LSHIFT()**
Implements in-place $\text{TOS} = \text{TOS1} \ll \text{TOS}$.

**INPLACE_RSHIFT()**
Implements in-place $\text{TOS} = \text{TOS1} \gg \text{TOS}$.

**INPLACE_AND()**
Implements in-place $\text{TOS} = \text{TOS1} \& \text{TOS}$.

**INPLACE_XOR()**
Implements in-place $\text{TOS} = \text{TOS1} \^ \text{TOS}$.
INPLACE_OR()
Implements in-place $\text{TOS} = \text{TOS1} \mid \text{TOS}$.

The slice opcodes take up to three parameters.

SLICE+0()
Implements $\text{TOS} = \text{TOS}[:]$.

SLICE+1()
Implements $\text{TOS} = \text{TOS1}[\text{TOS}:]$.

SLICE+2()
Implements $\text{TOS} = \text{TOS1}[:\text{TOS}]$.

SLICE+3()
Implements $\text{TOS} = \text{TOS2}[\text{TOS1}:\text{TOS}]$.

Slice assignment needs even an additional parameter. As any statement, they put nothing on the stack.

STORE_SLICE+0()
Implements $\text{TOS}[:] = \text{TOS1}$.

STORE_SLICE+1()
Implements $\text{TOS1}[\text{TOS}:] = \text{TOS2}$.

STORE_SLICE+2()
Implements $\text{TOS1}[:\text{TOS}] = \text{TOS2}$.

STORE_SLICE+3()
Implements $\text{TOS2}[\text{TOS1}:\text{TOS}] = \text{TOS3}$.

DELETE_SLICE+0()
Implements $\text{del} \ \text{TOS}[:]$. 
DELETE_SLICE+1()
  Implements del TOS1[TOS:].

DELETE_SLICE+2()
  Implements del TOS1[:TOS].

DELETE_SLICE+3()
  Implements del TOS2[TOS1:TOS].

STORE_SUBSCR()
  Implements TOS1[TOS] = TOS2.

DELETE_SUBSCR()
  Implements del TOS1[TOS].

Miscellaneous opcodes.

PRINT_EXPR()
  Implements the expression statement for the interactive mode. TOS is removed from the stack and printed. In non-interactive mode, an expression statement is terminated with POP_TOP.

PRINT_ITEM()
  Prints TOS to the file-like object bound to sys.stdout. There is one such instruction for each item in the print statement.

PRINT_ITEM_TO()
  Like PRINT_ITEM, but prints the item second from TOS to the file-like object at TOS. This is used by the extended print statement.

PRINT_NEWLINE()
  Prints a new line on sys.stdout. This is generated as the last operation of a print statement, unless the statement ends with a comma.
PRINT_NEWLINE_TO()

Like PRINT_NEWLINE, but prints the new line on the file-like object on the TOS. This is used by the extended print statement.

BREAK_LOOP()

Terminates a loop due to a break statement.

CONTINUE_LOOP(target)

Continues a loop due to a continue statement. target is the address to jump to (which should be a FOR_ITER instruction).

LIST_APPEND(i)

Calls list.append(TOS[-i], TOS). Used to implement list comprehensions. While the appended value is popped off, the list object remains on the stack so that it is available for further iterations of the loop.

LOAD_LOCALS()

Pushes a reference to the locals of the current scope on the stack. This is used in the code for a class definition: After the class body is evaluated, the locals are passed to the class definition.

RETURN_VALUE()

Returns with TOS to the caller of the function.

YIELD_VALUE()

Pops tos and yields it from a generator.

IMPORT_STAR()

Loads all symbols not starting with '_' directly from the module TOS to the local namespace. The module is popped after loading all names. This opcode implements from module import *.

EXEC_STMT()
Implements `exec TOS2,TOS1,TOS`. The compiler fills missing optional parameters with `None`.

**POP_BLOCK()**
Removes one block from the block stack. Per frame, there is a stack of blocks, denoting nested loops, try statements, and such.

**END_FINALLY()**
Terminates a `finally` clause. The interpreter recalls whether the exception has to be re-raised, or whether the function returns, and continues with the outer-next block.

**BUILD_CLASS()**
Creates a new class object. TOS is the methods dictionary, TOS1 the tuple of the names of the base classes, and TOS2 the class name.

**SETUP_WITH(delta)**
This opcode performs several operations before a with block starts. First, it loads `__exit__()` from the context manager and pushes it onto the stack for later use by `WITH_CLEANUP`. Then, `__enter__()` is called, and a finally block pointing to `delta` is pushed. Finally, the result of calling the enter method is pushed onto the stack. The next opcode will either ignore it (POP_TOP), or store it in (a) variable(s) (STORE_FAST, STORE_NAME, or UNPACK_SEQUENCE).

**WITH_CLEANUP()**
Cleans up the stack when a `with` statement block exits. On top of the stack are 1–3 values indicating how/why the finally clause was entered:

- TOP = `None`
- (TOP, SECOND) = (`WHY_{RETURN, CONTINUE}`), retval
- TOP = WHY_*; no retval below it
- (TOP, SECOND, THIRD) = exc_info()

Under them is EXIT, the context manager’s __exit__() bound method.

In the last case, EXIT(TOP, SECOND, THIRD) is called, otherwise EXIT(None, None, None).

EXIT is removed from the stack, leaving the values above it in the same order. In addition, if the stack represents an exception, and the function call returns a ‘true’ value, this information is "zapped", to prevent END_FINALLY from re-raising the exception. (But non-local gotos should still be resumed.)

All of the following opcodes expect arguments. An argument is two bytes, with the more significant byte last.

STORE_NAME(namei)
Implements name = TOS. namei is the index of name in the attribute co_names of the code object. The compiler tries to use STORE_FAST or STORE_GLOBAL if possible.

DELETE_NAME(namei)
Implements del name, where namei is the index into co_names attribute of the code object.

UNPACK_SEQUENCE(count)
Unpacks TOS into count individual values, which are put onto the stack right-to-left.

DUP_TOPX(count)
Duplicate count items, keeping them in the same order. Due to implementation limits, count should be between 1 and 5 inclusive.
STORE_ATTR(namei)
    Implements TOS\_name = TOS1, where namei is the index of name
    in co\_names.

DELETE_ATTR(namei)
    Implements del TOS\_name, using namei as index into co\_names.

STORE_GLOBAL(namei)
    Works as STORE\_NAME, but stores the name as a global.

DELETE_GLOBAL(namei)
    Works as DELETE\_NAME, but deletes a global name.

LOAD_CONST(consti)
    Pushes co\_consts[consti] onto the stack.

LOAD_NAME(namei)
    Pushes the value associated with co\_names[namei] onto the stack.

BUILD_TUPLE(count)
    Creates a tuple consuming count items from the stack, and
    pushes the resulting tuple onto the stack.

BUILD_LIST(count)
    Works as BUILD\_TUPLE, but creates a list.

BUILD_SET(count)
    Works as BUILD\_TUPLE, but creates a set.

    *New in version 2.7.*

BUILD_MAP(count)
    Pushes a new dictionary object onto the stack. The dictionary is
    pre-sized to hold count entries.
LOAD_ATTR(namei)
Replaces TOS with `getattr(TOS, co_names[namei])`.

COMPARE_OP(opname)
Performs a Boolean operation. The operation name can be found in `cmp_op[opname]`.

IMPORT_NAME(namei)
Imports the module `co_names[namei]`. TOS and TOS1 are popped and provide the `fromlist` and `level` arguments of `__import__()`.
The module object is pushed onto the stack. The current namespace is not affected: for a proper import statement, a subsequent `STORE_FAST` instruction modifies the namespace.

IMPORT_FROM(namei)
Loads the attribute `co_names[namei]` from the module found in TOS. The resulting object is pushed onto the stack, to be subsequently stored by a `STORE_FAST` instruction.

JUMP_FORWARD(delta)
Increments bytecode counter by `delta`.

POP_JUMP_IF_TRUE(target)
If TOS is true, sets the bytecode counter to `target`. TOS is popped.

POP_JUMP_IF_FALSE(target)
If TOS is false, sets the bytecode counter to `target`. TOS is popped.

JUMP_IF_TRUE_OR_POP(target)
If TOS is true, sets the bytecode counter to `target` and leaves TOS on the stack. Otherwise (TOS is false), TOS is popped.

JUMP_IF_FALSE_OR_POP(target)
If TOS is false, sets the bytecode counter to `target` and leaves TOS on the stack. Otherwise (TOS is true), TOS is popped.

**JUMP_ABSOLUTE**(*target*)

Set bytecode counter to `target`.

**FOR_ITER**(*delta*)

TOS is an *iterator*. Call its `next()` method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted TOS is popped, and the bytecode counter is incremented by `delta`.

**LOAD_GLOBAL**(*name*)

Loads the global named `co_names[name]` onto the stack.

**SETUP_LOOP**(*delta*)

Pushes a block for a loop onto the block stack. The block spans from the current instruction with a size of `delta` bytes.

**SETUP_EXCEPT**(*delta*)

Pushes a try block from a try-except clause onto the block stack. `delta` points to the first except block.

**SETUP_FINALLY**(*delta*)

Pushes a try block from a try-except clause onto the block stack. `delta` points to the finally block.

**STORE_MAP**()

Store a key and value pair in a dictionary. Pops the key and value while leaving the dictionary on the stack.

**LOAD_FAST**(*var_num*)

Pushes a reference to the local `co_varnames[var_num]` onto the stack.

**STORE_FAST**(*var_num*)
Stores TOS into the local `co_varnames[var_num]`.

**DELETE_FAST**(var_num)

Deletes local `co_varnames[var_num]`.

**LOAD_CLOSURE**(i)

Pushes a reference to the cell contained in slot `i` of the cell and free variable storage. The name of the variable is `co_cellvars[i]` if `i` is less than the length of `co_cellvars`. Otherwise it is `co_freevars[i - len(co_cellvars)]`.

**LOAD_DEREF**(i)

Loads the cell contained in slot `i` of the cell and free variable storage. Pushes a reference to the object the cell contains on the stack.

**STORE_DEREF**(i)

Stores TOS into the cell contained in slot `i` of the cell and free variable storage.

**SET_LINENO**(lineno)

This opcode is obsolete.

**RAISE_VARARGS**(argc)

Raises an exception. `argc` indicates the number of parameters to the raise statement, ranging from 0 to 3. The handler will find the traceback as TOS2, the parameter as TOS1, and the exception as TOS.

**CALL_FUNCTION**(argc)

Calls a function. The low byte of `argc` indicates the number of positional parameters, the high byte the number of keyword parameters. On the stack, the opcode finds the keyword parameters first. For each keyword argument, the value is on top of the key. Below the keyword parameters, the positional
parameters are on the stack, with the right-most parameter on top. Below the parameters, the function object to call is on the stack. Pops all function arguments, and the function itself off the stack, and pushes the return value.

**MAKE_FUNCTION**(argc)

Pushes a new function object on the stack. TOS is the code associated with the function. The function object is defined to have `argc` default parameters, which are found below TOS.

**MAKE_CLOSURE**(argc)

Creates a new function object, sets its `func_closure` slot, and pushes it on the stack. TOS is the code associated with the function, TOS1 the tuple containing cells for the closure's free variables. The function also has `argc` default parameters, which are found below the cells.

**BUILD_SLICE**(argc)

Pushes a slice object on the stack. `argc` must be 2 or 3. If it is 2, `slice(TOS1, TOS)` is pushed; if it is 3, `slice(TOS2, TOS1, TOS)` is pushed. See the `slice()` built-in function for more information.

**EXTENDED_ARG**(ext)

Prefixes any opcode which has an argument too big to fit into the default two bytes. `ext` holds two additional bytes which, taken together with the subsequent opcode’s argument, comprise a four-byte argument, `ext` being the two most-significant bytes.

**CALL_FUNCTION_VAR**(argc)

Calls a function. `argc` is interpreted as in `CALL_FUNCTION`. The top element on the stack contains the variable argument list, followed by keyword and positional arguments.

**CALL_FUNCTION_KW**(argc)

Calls a function. `argc` is interpreted as in `CALL_FUNCTION`. The top
element on the stack contains the keyword arguments dictionary, followed by explicit keyword and positional arguments.

**CALL_FUNCTION_VAR_Kw(argc)**

Calls a function. *argc* is interpreted as in `CALL_FUNCTION`. The top element on the stack contains the keyword arguments dictionary, followed by the variable-arguments tuple, followed by explicit keyword and positional arguments.

**HAVE_ARGUMENT()**

This is not really an opcode. It identifies the dividing line between opcodes which don't take arguments `< HAVE_ARGUMENT` and those which do `>= HAVE_ARGUMENT`.
32.13. pickletools — Tools for pickle developers

New in version 2.3.

Source code: Lib.pickletools.py

This module contains various constants relating to the intimate details of the pickle module, some lengthy comments about the implementation, and a few useful functions for analyzing pickled data. The contents of this module are useful for Python core developers who are working on the pickle and cPickle implementations; ordinary users of the pickle module probably won’t find the pickletools module relevant.

pickletools.dis(pickle, out=None, memo=None, indentlevel=4)
Outputs a symbolic disassembly of the pickle to the file-like object out, defaulting to sys.stdout. pickle can be a string or a file-like object. memo can be a Python dictionary that will be used as the pickle’s memo; it can be used to perform disassemblies across multiple pickles created by the same pickler. Successive levels, indicated by MARK opcodes in the stream, are indented by indentlevel spaces.

pickletools.genops(pickle)
Provides an iterator over all of the opcodes in a pickle, returning a sequence of (opcode, arg, pos) triples. opcode is an instance of an OpcodeInfo class; arg is the decoded value, as a Python object, of the opcode’s argument; pos is the position at which this opcode is located. pickle can be a string or a file-like object.

pickletools.optimize(picklestring)
Returns a new equivalent pickle string after eliminating unused `PUT` opcodes. The optimized pickle is shorter, takes less transmission time, requires less storage space, and unpickles more efficiently.

*New in version 2.6.*
33. Python compiler package

**Deprecated since version 2.6:** The `compiler` package has been removed in Python 3.

The Python compiler package is a tool for analyzing Python source code and generating Python bytecode. The compiler contains libraries to generate an abstract syntax tree from Python source code and to generate Python bytecode from the tree.

The `compiler` package is a Python source to bytecode translator written in Python. It uses the built-in parser and standard `parser` module to generate a concrete syntax tree. This tree is used to generate an abstract syntax tree (AST) and then Python bytecode.

The full functionality of the package duplicates the built-in compiler provided with the Python interpreter. It is intended to match its behavior almost exactly. Why implement another compiler that does the same thing? The package is useful for a variety of purposes. It can be modified more easily than the built-in compiler. The AST it generates is useful for analyzing Python source code.

This chapter explains how the various components of the `compiler` package work. It blends reference material with a tutorial.
33.1. The basic interface

The top-level of the package defines four functions. If you import *compiler*, you will get these functions and a collection of modules contained in the package.

**compiler.parse(buf)**

Returns an abstract syntax tree for the Python source code in *buf*. The function raises *SyntaxError* if there is an error in the source code. The return value is a *compiler.ast.Module* instance that contains the tree.

**compiler.parseFile(path)**

Return an abstract syntax tree for the Python source code in the file specified by *path*. It is equivalent to `parse(open(path).read())`.

**compiler.walk(ast, visitor[, verbose])**

Do a pre-order walk over the abstract syntax tree *ast*. Call the appropriate method on the *visitor* instance for each node encountered.

**compiler.compile(source, filename, mode, flags=None, dont_inherit=None)**

Compile the string *source*, a Python module, statement or expression, into a code object that can be executed by the exec statement or *eval().* This function is a replacement for the built-in *compile()* function.

The *filename* will be used for run-time error messages.

The *mode* must be ‘exec’ to compile a module, ‘single’ to compile a single (interactive) statement, or ‘eval’ to compile an
expression.

The flags and dont_inherit arguments affect future-related statements, but are not supported yet.

```
compiler.compileFile(source)
```

Compiles the file source and generates a .pyc file.

The compiler package contains the following modules: ast, consts, future, misc, pyassem, pycodegen, symbols, transformer, and visitor.
33.2. Limitations

There are some problems with the error checking of the compiler package. The interpreter detects syntax errors in two distinct phases. One set of errors is detected by the interpreter’s parser, the other set by the compiler. The compiler package relies on the interpreter’s parser, so it gets the first phases of error checking for free. It implements the second phase itself, and that implementation is incomplete. For example, the compiler package does not raise an error if a name appears more than once in an argument list: `def f(x, x): ...`

A future version of the compiler should fix these problems.
33.3. Python Abstract Syntax

The \texttt{compiler.ast} module defines an abstract syntax for Python. In the abstract syntax tree, each node represents a syntactic construct. The root of the tree is \texttt{Module} object.

The abstract syntax offers a higher level interface to parsed Python source code. The \texttt{parser} module and the compiler written in C for the Python interpreter use a concrete syntax tree. The concrete syntax is tied closely to the grammar description used for the Python parser. Instead of a single node for a construct, there are often several levels of nested nodes that are introduced by Python’s precedence rules.

The abstract syntax tree is created by the \texttt{compiler.transformer} module. The transformer relies on the built-in Python parser to generate a concrete syntax tree. It generates an abstract syntax tree from the concrete tree.

The \texttt{transformer} module was created by Greg Stein and Bill Tutt for an experimental Python-to-C compiler. The current version contains a number of modifications and improvements, but the basic form of the abstract syntax and of the transformer are due to Stein and Tutt.

33.3.1. AST Nodes

The \texttt{compiler.ast} module is generated from a text file that describes each node type and its elements. Each node type is represented as a class that inherits from the abstract base class \texttt{compiler.ast.Node} and defines a set of named attributes for child nodes.

\texttt{class compiler.ast.Node}

The \texttt{Node} instances are created automatically by the parser
generator. The recommended interface for specific Node instances is to use the public attributes to access child nodes. A public attribute may be bound to a single node or to a sequence of nodes, depending on the Node type. For example, the bases attribute of the class node, is bound to a list of base class nodes, and the doc attribute is bound to a single node.

Each Node instance has a lineno attribute which may be None. XXX Not sure what the rules are for which nodes will have a useful lineno.

All Node objects offer the following methods:

getChildren()

Returns a flattened list of the child nodes and objects in the order they occur. Specifically, the order of the nodes is the order in which they appear in the Python grammar. Not all of the children are Node instances. The names of functions and classes, for example, are plain strings.

getChildNodes()

Returns a flattened list of the child nodes in the order they occur. This method is like getChildren(), except that it only returns those children that are Node instances.

Two examples illustrate the general structure of Node classes. The while statement is defined by the following grammar production:

```
while_stmt:    "while" expression ":" suite
              ["else" ":" suite]
```

The While node has three attributes: test, body, and else_. (If the natural name for an attribute is also a Python reserved word, it can't be used as an attribute name. An underscore is appended to the
word to make it a legal identifier, hence `else_` instead of `else`.)

The `if` statement is more complicated because it can include several tests.

```
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
```

The `If` node only defines two attributes: `tests` and `else_`. The `tests` attribute is a sequence of test expression, consequent body pairs. There is one pair for each `if/elif` clause. The first element of the pair is the test expression. The second elements is a `stmt` node that contains the code to execute if the test is true.

The `getChildren()` method of `If` returns a flat list of child nodes. If there are three `if/elif` clauses and no `else` clause, then `getChildren()` will return a list of six elements: the first test expression, the first `stmt`, the second test expression, etc.

The following table lists each of the `Node` subclasses defined in `compiler.ast` and each of the public attributes available on their instances. The values of most of the attributes are themselves `Node` instances or sequences of instances. When the value is something other than an instance, the type is noted in the comment. The attributes are listed in the order in which they are returned by `getChildren()` and `getChildNodes()`.

<table>
<thead>
<tr>
<th>Node type</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td><code>left</code></td>
<td>left operand</td>
</tr>
<tr>
<td></td>
<td><code>right</code></td>
<td>right operand</td>
</tr>
<tr>
<td>And</td>
<td><code>nodes</code></td>
<td>list of operands</td>
</tr>
<tr>
<td>AssAttr</td>
<td></td>
<td><code>attribute as target of assignment</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>expression on the left-hand</td>
</tr>
<tr>
<td>Token</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><code>expr</code></td>
<td>side of the dot</td>
<td></td>
</tr>
<tr>
<td><code>attrname</code></td>
<td>the attribute name, a string</td>
<td></td>
</tr>
<tr>
<td><code>flags</code></td>
<td>XXX</td>
<td></td>
</tr>
<tr>
<td><code>AssList</code></td>
<td>nodes: list of list elements being assigned to</td>
<td></td>
</tr>
<tr>
<td><code>AssName</code></td>
<td>name: name being assigned to</td>
<td></td>
</tr>
<tr>
<td><code>flags</code></td>
<td>XXX</td>
<td></td>
</tr>
<tr>
<td><code>AssTuple</code></td>
<td>nodes: list of tuple elements being assigned to</td>
<td></td>
</tr>
<tr>
<td><code>Assert</code></td>
<td><code>test</code>: the expression to be tested</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>fail</code>: the value of the <code>AssertionError</code></td>
<td></td>
</tr>
<tr>
<td><code>Assign</code></td>
<td>nodes: a list of assignment targets, one per equal sign</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>expr</code>: the value being assigned</td>
<td></td>
</tr>
<tr>
<td><code>AugAssign</code></td>
<td>node</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>op</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>expr</code></td>
<td></td>
</tr>
<tr>
<td><code>Backquote</code></td>
<td><code>expr</code></td>
<td></td>
</tr>
<tr>
<td><code>Bitand</code></td>
<td>nodes</td>
<td></td>
</tr>
<tr>
<td><code>Bitor</code></td>
<td>nodes</td>
<td></td>
</tr>
<tr>
<td><code>Bitxor</code></td>
<td>nodes</td>
<td></td>
</tr>
<tr>
<td><code>Break</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>CallFunc</code></td>
<td>node: expression for the callee</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>args</code>: a list of arguments</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>star_args</code>: the extended *-arg value</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>dstar_args</code>: the extended **-arg value</td>
<td></td>
</tr>
<tr>
<td><code>Class</code></td>
<td><code>name</code>: the name of the class, a string</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>bases</code>: a list of base classes</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>doc</code>: doc string, a string or <code>None</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>code</code>: the body of the class</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>decorators</td>
<td>DOCtors of None</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
<td>-----------------</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>name used in def, a string</td>
</tr>
<tr>
<td></td>
<td>argnames</td>
<td>list of argument names, as strings</td>
</tr>
<tr>
<td></td>
<td>defaults</td>
<td>list of default values</td>
</tr>
<tr>
<td></td>
<td>flags</td>
<td>XXX</td>
</tr>
<tr>
<td></td>
<td>doc</td>
<td>doc string, a string or None</td>
</tr>
<tr>
<td></td>
<td>code</td>
<td></td>
</tr>
</tbody>
</table>

| node        | | |
| expr        | | |
| ops         | | |
| value       | | |
| node        | | |
| locas       | | |
| globals     | | |
| left        | | |
| right       | | |
| list        | | |
| body        | | |
| else_       | | |
| modname     | | |
| names       | | |

| expr        | | |
| left        | | |
| right       | | |
| statement   | | |

| Const       | | |
| statement   | | |

| Const       | | |
| statement   | | |

| Const       | | |
| statement   | | |

| Continue    | | |
| statement   | | |

| Decorators  | nodes | List of function decorator expressions |
| Dict        | items  | |
| Decorators  | nodes  | |
| Dict        | items  | |
| Discard     | expr   | |
| Div         | left    | |
| right       | | |
| Ellipsis    | | |
| Expression  | node   | |
| Exec        | expr   | |
| locals      | | |
| globals     | | |
| FloorDiv    | left   | |
| right       | | |
| For         | assign | |
| list        | | |
| body        | | |
| else_       | | |
| From        | modname| |
| names       | | |
| Function    | decorators | Decorators of None |
|             | name       | name used in def, a string |
|             | argnames   | list of argument names, as strings |
|             | defaults   | list of default values |
|             | flags      | XXX |
|             | doc        | doc string, a string or None |
|             | code       | |

| Const       | | |
| statement   | | |

| Const       | | |
| statement   | | |

| Const       | | |
| statement   | | |

| Const       | | |
| statement   | | |

| Continue    | | |
| statement   | | |

<p>| Decorators  | nodes | List of function decorator expressions |
| Dict        | items  | |
| Decorators  | nodes  | |
| Dict        | items  | |
| Discard     | expr   | |
| Div         | left    | |
| right       | | |
| Ellipsis    | | |
| Expression  | node   | |
| Exec        | expr   | |
| locals      | | |
| globals     | | |
| FloorDiv    | left   | |
| right       | | |
| For         | assign | |
| list        | | |
| body        | | |
| else_       | | |
| From        | modname| |
| names       | | |
| Function    | decorators | Decorators of None |
|             | name       | name used in def, a string |
|             | argnames   | list of argument names, as strings |
|             | defaults   | list of default values |
|             | flags      | XXX |
|             | doc        | doc string, a string or None |
|             | code       | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GenExpr</td>
<td>code</td>
</tr>
<tr>
<td>GenExprFor</td>
<td>assign</td>
</tr>
<tr>
<td></td>
<td>iter</td>
</tr>
<tr>
<td></td>
<td>ifs</td>
</tr>
<tr>
<td>GenExprIf</td>
<td>test</td>
</tr>
<tr>
<td>GenExprInner</td>
<td>expr</td>
</tr>
<tr>
<td></td>
<td>qals</td>
</tr>
<tr>
<td>Getattr</td>
<td>expr</td>
</tr>
<tr>
<td></td>
<td>attrname</td>
</tr>
<tr>
<td>Global</td>
<td>names</td>
</tr>
<tr>
<td>If</td>
<td>tests</td>
</tr>
<tr>
<td></td>
<td>else_</td>
</tr>
<tr>
<td>Import</td>
<td>names</td>
</tr>
<tr>
<td>Invert</td>
<td>expr</td>
</tr>
<tr>
<td>Keyword</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>expr</td>
</tr>
<tr>
<td>Lambda</td>
<td>argnames</td>
</tr>
<tr>
<td></td>
<td>defaults</td>
</tr>
<tr>
<td></td>
<td>flags</td>
</tr>
<tr>
<td></td>
<td>code</td>
</tr>
<tr>
<td>LeftShift</td>
<td>left</td>
</tr>
<tr>
<td></td>
<td>right</td>
</tr>
<tr>
<td>List</td>
<td>nodes</td>
</tr>
<tr>
<td>ListComp</td>
<td>expr</td>
</tr>
<tr>
<td></td>
<td>qals</td>
</tr>
<tr>
<td>ListCompFor</td>
<td>assign</td>
</tr>
<tr>
<td></td>
<td>list</td>
</tr>
<tr>
<td></td>
<td>ifs</td>
</tr>
<tr>
<td>ListCompIf</td>
<td>test</td>
</tr>
<tr>
<td>Mod</td>
<td>left</td>
</tr>
<tr>
<td></td>
<td>right</td>
</tr>
<tr>
<td>Module</td>
<td>doc</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td>node</td>
</tr>
<tr>
<td>Mul</td>
<td>left</td>
</tr>
<tr>
<td></td>
<td>right</td>
</tr>
<tr>
<td>Name</td>
<td>name</td>
</tr>
<tr>
<td>Not</td>
<td>expr</td>
</tr>
<tr>
<td>Or</td>
<td>nodes</td>
</tr>
<tr>
<td>Pass</td>
<td></td>
</tr>
<tr>
<td>Power</td>
<td>left</td>
</tr>
<tr>
<td></td>
<td>right</td>
</tr>
<tr>
<td>Print</td>
<td>nodes</td>
</tr>
<tr>
<td></td>
<td>dest</td>
</tr>
<tr>
<td>Printnl</td>
<td>nodes</td>
</tr>
<tr>
<td></td>
<td>dest</td>
</tr>
<tr>
<td>Raise</td>
<td>expr1</td>
</tr>
<tr>
<td></td>
<td>expr2</td>
</tr>
<tr>
<td></td>
<td>expr3</td>
</tr>
<tr>
<td>Return</td>
<td>value</td>
</tr>
<tr>
<td>RightShift</td>
<td>left</td>
</tr>
<tr>
<td></td>
<td>right</td>
</tr>
<tr>
<td>Slice</td>
<td>expr</td>
</tr>
<tr>
<td></td>
<td>flags</td>
</tr>
<tr>
<td></td>
<td>lower</td>
</tr>
<tr>
<td></td>
<td>upper</td>
</tr>
<tr>
<td>Sliceobj</td>
<td>nodes</td>
</tr>
<tr>
<td>Stmt</td>
<td>nodes</td>
</tr>
<tr>
<td>Sub</td>
<td>left</td>
</tr>
<tr>
<td></td>
<td>right</td>
</tr>
<tr>
<td>Subscript</td>
<td>expr</td>
</tr>
<tr>
<td></td>
<td>flags</td>
</tr>
<tr>
<td></td>
<td>subs</td>
</tr>
</tbody>
</table>
33.3.2. Assignment nodes

There is a collection of nodes used to represent assignments. Each assignment statement in the source code becomes a single `Assign` node in the AST. The `nodes` attribute is a list that contains a node for each assignment target. This is necessary because assignment can be chained, e.g. `a = b = 2`. Each `Node` in the list will be one of the following classes: `AssAttr`, `AssList`, `AssName`, or `AssTuple`.

Each target assignment node will describe the kind of object being assigned to: `AssName` for a simple name, e.g. `a = 1`. `AssAttr` for an attribute assigned, e.g. `a.x = 1`. `AssList` and `AssTuple` for list and tuple expansion respectively, e.g. `a, b, c = a_tuple`.

The target assignment nodes also have a `flags` attribute that indicates whether the node is being used for assignment or in a
delete statement. The AssName is also used to represent a delete statement, e.g. del x.

When an expression contains several attribute references, an assignment or delete statement will contain only one AssAttr node – for the final attribute reference. The other attribute references will be represented as Getattr nodes in the expr attribute of the AssAttr instance.

### 33.3.3. Examples

This section shows several simple examples of ASTs for Python source code. The examples demonstrate how to use the parse() function, what the repr of an AST looks like, and how to access attributes of an AST node.

The first module defines a single function. Assume it is stored in doublelib.py.

```python
"""This is an example module.

This is the docstring.
"""

def double(x):
    "Return twice the argument"
    return x * 2
```

In the interactive interpreter session below, I have reformatted the long AST reprs for readability. The AST reprs use unqualified class names. If you want to create an instance from a repr, you must import the class names from the compiler.ast module.

```python
>>> import compiler
>>> mod = compiler.parseFile("doublelib.py")
>>> mod
Module('This is an example module.

This is the docstring.
',

def double(x):
    "Return twice the argument"
    return x * 2
```
>>> from compiler.ast import *
>>> Module('This is an example module.

This is the docstring.

Function(None, 'double', ['x'], [], 0, 'Return twice the argument',
Stmt([Return(Mul((Name('x'), Const(2))))]))

>>> mod.doc
'This is an example module.

This is the docstring.
'
>>> for node in mod.node.nodes:
    print node

Function(None, 'double', ['x'], [], 0, 'Return twice the argument',
Stmt([Return(Mul((Name('x'), Const(2))))]))

>>> func = mod.node.nodes[0]
>>> func.code
Stmt([Return(Mul((Name('x'), Const(2))))])
33.4. Using Visitors to Walk ASTs

The visitor pattern is ... The compiler package uses a variant on the visitor pattern that takes advantage of Python’s introspection features to eliminate the need for much of the visitor’s infrastructure.

The classes being visited do not need to be programmed to accept visitors. The visitor need only define visit methods for classes it is specifically interested in; a default visit method can handle the rest.

XXX The magic visit() method for visitors.

```
compiler.visitor.walk(tree, visitor[, verbose])
```

```
class compiler.visitor.ASTVisitor
```

The ASTVisitor is responsible for walking over the tree in the correct order. A walk begins with a call to preorder(). For each node, it checks the visitor argument to preorder() for a method named ‘visitNodeType,’ where NodeType is the name of the node’s class, e.g. for a While node a visitWhile() would be called. If the method exists, it is called with the node as its first argument.

The visitor method for a particular node type can control how child nodes are visited during the walk. The ASTVisitor modifies the visitor argument by adding a visit method to the visitor; this method can be used to visit a particular child node. If no visitor is found for a particular node type, the default() method is called.

ASTVisitor objects have the following methods:

XXX describe extra arguments
default(node[,...])

dispatch(node[,...])

preorder(tree, visitor)
33.5. Bytecode Generation

The code generator is a visitor that emits bytecodes. Each visit method can call the `emit()` method to emit a new bytecode. The basic code generator is specialized for modules, classes, and functions. An assembler converts that emitted instructions to the low-level bytecode format. It handles things like generation of constant lists of code objects and calculation of jump offsets.
34. Miscellaneous Services

The modules described in this chapter provide miscellaneous services that are available in all Python versions. Here's an overview:

- **34.1. formatter — Generic output formatting**
  - 34.1.1. The Formatter Interface
  - 34.1.2. Formatter Implementations
  - 34.1.3. The Writer Interface
  - 34.1.4. Writer Implementations
34.1. **formatter — Generic output formatting**

This module supports two interface definitions, each with multiple implementations. The *formatter* interface is used by the `HTMLParser` class of the `htmllib` module, and the *writer* interface is required by the formatter interface.

Formatter objects transform an abstract flow of formatting events into specific output events on writer objects. Formatters manage several stack structures to allow various properties of a writer object to be changed and restored; writers need not be able to handle relative changes nor any sort of “change back” operation. Specific writer properties which may be controlled via formatter objects are horizontal alignment, font, and left margin indentations. A mechanism is provided which supports providing arbitrary, non-exclusive style settings to a writer as well. Additional interfaces facilitate formatting events which are not reversible, such as paragraph separation.

Writer objects encapsulate device interfaces. Abstract devices, such as file formats, are supported as well as physical devices. The provided implementations all work with abstract devices. The interface makes available mechanisms for setting the properties which formatter objects manage and inserting data into the output.
34.1.1. The Formatter Interface

Interfaces to create formatters are dependent on the specific formatter class being instantiated. The interfaces described below are the required interfaces which all formatters must support once initialized.

One data element is defined at the module level:

`formatter.AS_IS`

Value which can be used in the font specification passed to the `push_font()` method described below, or as the new value to any other `push_property()` method. Pushing the `AS_IS` value allows the corresponding `pop_property()` method to be called without having to track whether the property was changed.

The following attributes are defined for formatter instance objects:

`formatter.writer`

The writer instance with which the formatter interacts.

`formatter.end_paragraph(blanklines)`

Close any open paragraphs and insert at least `blanklines` before the next paragraph.

`formatter.add_line_break()`

Add a hard line break if one does not already exist. This does not break the logical paragraph.

`formatter.add_hor_rule(*args, **kw)`

Insert a horizontal rule in the output. A hard break is inserted if there is data in the current paragraph, but the logical paragraph is not broken. The arguments and keywords are passed on to the writer’s `send_line_break()` method.
formatter.\texttt{add\_flowing\_data}(\textit{data})

Provide data which should be formatted with collapsed whitespace. Whitespace from preceding and successive calls to \texttt{add\_flowing\_data()} is considered as well when the whitespace collapse is performed. The data which is passed to this method is expected to be word-wrapped by the output device. Note that any word-wrapping still must be performed by the writer object due to the need to rely on device and font information.

formatter.\texttt{add\_literal\_data}(\textit{data})

Provide data which should be passed to the writer unchanged. Whitespace, including newline and tab characters, are considered legal in the value of \textit{data}.

formatter.\texttt{add\_label\_data}(\textit{format}, \textit{counter})

Insert a label which should be placed to the left of the current left margin. This should be used for constructing bulleted or numbered lists. If the \textit{format} value is a string, it is interpreted as a format specification for \textit{counter}, which should be an integer. The result of this formatting becomes the value of the label; if \textit{format} is not a string it is used as the label value directly. The label value is passed as the only argument to the writer's \texttt{send\_label\_data()} method. Interpretation of non-string label values is dependent on the associated writer.

Format specifications are strings which, in combination with a counter value, are used to compute label values. Each character in the format string is copied to the label value, with some characters recognized to indicate a transform on the counter value. Specifically, the character '1' represents the counter value formatter as an Arabic number, the characters 'A' and 'a' represent alphabetic representations of the counter value in upper and lower case, respectively, and 'I' and 'i' represent the counter value in Roman numerals, in upper and lower case.
Note that the alphabetic and roman transforms require that the counter value be greater than zero.

`formatter.flush_softspace()`
Send any pending whitespace buffered from a previous call to `add_flowing_data()` to the associated writer object. This should be called before any direct manipulation of the writer object.

`formatter.push_alignment(align)`
Push a new alignment setting onto the alignment stack. This may be `AS_IS` if no change is desired. If the alignment value is changed from the previous setting, the writer’s `new_alignment()` method is called with the `align` value.

`formatter.pop_alignment()`
Restore the previous alignment.

`formatter.push_font((size, italic, bold, teletype))`
Change some or all font properties of the writer object. Properties which are not set to `AS_IS` are set to the values passed in while others are maintained at their current settings. The writer’s `new_font()` method is called with the fully resolved font specification.

`formatter.pop_font()`
Restore the previous font.

`formatter.push_margin(margin)`
Increase the number of left margin indentations by one, associating the logical tag `margin` with the new indentation. The initial margin level is 0. Changed values of the logical tag must be true values; false values other than `AS_IS` are not sufficient to change the margin.

`formatter.pop_margin()`
Restore the previous margin.

**formatter.push_style(*styles)**

Push any number of arbitrary style specifications. All styles are pushed onto the styles stack in order. A tuple representing the entire stack, including `AS_IS` values, is passed to the writer’s `new_styles()` method.

**formatter.pop_style([n=1])**

Pop the last n style specifications passed to `push_style()`. A tuple representing the revised stack, including `AS_IS` values, is passed to the writer’s `new_styles()` method.

**formatter.set_spacing(spacing)**

Set the spacing style for the writer.

**formatter.assert_line_data([flag=1])**

Inform the formatter that data has been added to the current paragraph out-of-band. This should be used when the writer has been manipulated directly. The optional `flag` argument can be set to false if the writer manipulations produced a hard line break at the end of the output.
34.1.2. Formatter Implementations

Two implementations of formatter objects are provided by this module. Most applications may use one of these classes without modification or subclassing.

```python
class formatter.NullFormatter([writer])
    A formatter which does nothing. If writer is omitted, a NullWriter instance is created. No methods of the writer are called by NullFormatter instances. Implementations should inherit from this class if implementing a writer interface but don’t need to inherit any implementation.
```

```python
class formatter.AbstractFormatter(writer)
    The standard formatter. This implementation has demonstrated wide applicability to many writers, and may be used directly in most circumstances. It has been used to implement a full-featured World Wide Web browser.
```
34.1.3. The Writer Interface

Interfaces to create writers are dependent on the specific writer class being instantiated. The interfaces described below are the required interfaces which all writers must support once initialized. Note that while most applications can use the AbstractFormatter class as a formatter, the writer must typically be provided by the application.

writer.**flush**()
Flush any buffered output or device control events.

writer.**new_alignment**(align)
Set the alignment style. The align value can be any object, but by convention is a string or None, where None indicates that the writer’s “preferred” alignment should be used. Conventional align values are 'left', 'center', 'right', and 'justify'.

writer.**new_font**(font)
Set the font style. The value of font will be None, indicating that the device’s default font should be used, or a tuple of the form (size, italic, bold, teletype). Size will be a string indicating the size of font that should be used; specific strings and their interpretation must be defined by the application. The italic, bold, and teletype values are Boolean values specifying which of those font attributes should be used.

writer.**new_margin**(margin, level)
Set the margin level to the integer level and the logical tag to margin. Interpretation of the logical tag is at the writer’s discretion; the only restriction on the value of the logical tag is that it not be a false value for non-zero values of level.

writer.**new_spacing**(spacing)
Set the spacing style to *spacing*.

**writer.** `new_styles(styles)`

Set additional styles. The *styles* value is a tuple of arbitrary values; the value `AS_IS` should be ignored. The *styles* tuple may be interpreted either as a set or as a stack depending on the requirements of the application and writer implementation.

**writer.** `send_line_break()`

Break the current line.

**writer.** `send_paragraph(blankline)`

Produce a paragraph separation of at least *blankline* blank lines, or the equivalent. The *blankline* value will be an integer. Note that the implementation will receive a call to `send_line_break()` before this call if a line break is needed; this method should not include ending the last line of the paragraph. It is only responsible for vertical spacing between paragraphs.

**writer.** `send_hor_rule(*args, **kw)`

Display a horizontal rule on the output device. The arguments to this method are entirely application- and writer-specific, and should be interpreted with care. The method implementation may assume that a line break has already been issued via `send_line_break()`.

**writer.** `send_flowing_data(data)`

Output character data which may be word-wrapped and re-flowed as needed. Within any sequence of calls to this method, the writer may assume that spans of multiple whitespace characters have been collapsed to single space characters.

**writer.** `send_literal_data(data)`

Output character data which has already been formatted for display. Generally, this should be interpreted to mean that line.
breaks indicated by newline characters should be preserved and no new line breaks should be introduced. The data may contain embedded newline and tab characters, unlike data provided to the `send_formatted_data()` interface.

`writer.send_label_data(data)`

Set `data` to the left of the current left margin, if possible. The value of `data` is not restricted; treatment of non-string values is entirely application- and writer-dependent. This method will only be called at the beginning of a line.
34.1.4. Writer Implementations

Three implementations of the writer object interface are provided as examples by this module. Most applications will need to derive new writer classes from the NullWriter class.

```python
class formatter.NullWriter
    A writer which only provides the interface definition; no actions are taken on any methods. This should be the base class for all writers which do not need to inherit any implementation methods.

class formatter.AbstractWriter
    A writer which can be used in debugging formatters, but not much else. Each method simply announces itself by printing its name and arguments on standard output.

class formatter.DumbWriter(file=None, maxcol=72)
    Simple writer class which writes output on the file object passed in as file or, if file is None, on standard output. The output is simply word-wrapped to the number of columns specified by maxcol. This class is suitable for reflowing a sequence of paragraphs.
```
This chapter describes modules that are only available on MS Windows platforms.

- **35.1. msilib** — Read and write Microsoft Installer files
  - 35.1.1. Database Objects
  - 35.1.2. View Objects
  - 35.1.3. Summary Information Objects
  - 35.1.4. Record Objects
  - 35.1.5. Errors
  - 35.1.6. CAB Objects
  - 35.1.7. Directory Objects
  - 35.1.8. Features
  - 35.1.9. GUI classes
  - 35.1.10. Precomputed tables
- **35.2. msvcrt** – Useful routines from the MS VC++ runtime
  - 35.2.1. File Operations
  - 35.2.2. Console I/O
  - 35.2.3. Other Functions
- **35.3. _winreg** – Windows registry access
  - 35.3.1. Constants
    - 35.3.1.1. HKEY_* Constants
    - 35.3.1.2. Access Rights
      - 35.3.1.2.1. 64-bit Specific
    - 35.3.1.3. Value Types
  - 35.3.2. Registry Handle Objects
- **35.4. winsound** — Sound-playing interface for Windows
35.1. msilib — Read and write Microsoft Installer files

*New in version 2.5.*

The `msilib` supports the creation of Microsoft Installer (.msi) files. Because these files often contain an embedded “cabinet” file (.cab), it also exposes an API to create CAB files. Support for reading .cab files is currently not implemented; read support for the .msi database is possible.

This package aims to provide complete access to all tables in an .msi file, therefore, it is a fairly low-level API. Two primary applications of this package are the `distutils` command `bdist_msi`, and the creation of Python installer package itself (although that currently uses a different version of `msilib`).

The package contents can be roughly split into four parts: low-level CAB routines, low-level MSI routines, higher-level MSI routines, and standard table structures.

```python
msilib.FCICreate(cabname, files)
```

Create a new CAB file named `cabname`. `files` must be a list of tuples, each containing the name of the file on disk, and the name of the file inside the CAB file.

The files are added to the CAB file in the order they appear in the list. All files are added into a single CAB file, using the MSZIP compression algorithm.

Callbacks to Python for the various steps of MSI creation are currently not exposed.
msilib. **UuidCreate()**

Return the string representation of a new unique identifier. This wraps the Windows API functions `UuidCreate()` and `UuidToString()`.

msilib. **OpenDatabase(path, persist)**

Return a new database object by calling MsiOpenDatabase. *path* is the file name of the MSI file; *persist* can be one of the constants `MSIDBOPEN_CREATEDIRECT`, `MSIDBOPEN_CREATE`, `MSIDBOPEN_DIRECT`, `MSIDBOPEN_READONLY`, or `MSIDBOPEN_TRANSACT`, and may include the flag `MSIDBOPEN_PATCHFILE`. See the Microsoft documentation for the meaning of these flags; depending on the flags, an existing database is opened, or a new one created.

msilib. **CreateRecord(count)**

Return a new record object by calling `MSICreateRecord()`. *count* is the number of fields of the record.

msilib. **init_database(name, schema, ProductName, ProductCode, ProductVersion, Manufacturer)**

Create and return a new database *name*, initialize it with *schema*, and set the properties *ProductName*, *ProductCode*, *ProductVersion*, and *Manufacturer*.

*schema* must be a module object containing tables and _Validation_records attributes; typically, `msilib.schema` should be used.

The database will contain just the schema and the validation records when this function returns.

msilib. **add_data(database, table, records)**

Add all *records* to the table named *table* in *database*.

The *table* argument must be one of the predefined tables in the
MSI schema, e.g. 'Feature', 'File', 'Component', 'Dialog', 'Control', etc.

*records* should be a list of tuples, each one containing all fields of a record according to the schema of the table. For optional fields, *None* can be passed.

Field values can be int or long numbers, strings, or instances of the Binary class.

```python
class msilib.Binary(filename)
    Represents entries in the Binary table; inserting such an object using add_data() reads the file named *filename* into the table.
```

```python
msilib.add_tables(database, module)
    Add all table content from *module* to *database*. *module* must contain an attribute *tables* listing all tables for which content should be added, and one attribute per table that has the actual content.

    This is typically used to install the sequence tables.
```

```python
msilib.add_stream(database, name, path)
    Add the file *path* into the _Stream table of *database*, with the stream name *name*.
```

```python
msilib.gen_uuid()
    Return a new UUID, in the format that MSI typically requires (i.e. in curly braces, and with all hexdigits in upper-case).
```

**See also:** FCICreateFile UuidCreate UuidToString
35.1.1. Database Objects

Database. **OpenView(sql)**
Return a view object, by calling **MSIDatabaseOpenView()**. *sql* is the SQL statement to execute.

Database. **Commit()**
Commit the changes pending in the current transaction, by calling **MSIDatabaseCommit()**.

Database. **GetSummaryInformation(count)**
Return a new summary information object, by calling **MsiGetSummaryInformation()**. *count* is the maximum number of updated values.

**See also:**  **MSIDatabaseOpenView**  **MSIDatabaseCommit**  **MsiGetSummaryInformation**
35.1.2. View Objects

View.**Execute**(params)

Execute the SQL query of the view, through MSIViewExecute(). If *params* is not None, it is a record describing actual values of the parameter tokens in the query.

View.**GetColumnInfo**(kind)

Return a record describing the columns of the view, through calling MsiViewGetColumnInfo(). *kind* can be either MSICOLINFO_NAMES or MSICOLINFO_TYPES.

View.**Fetch**()

Return a result record of the query, through calling MsiViewFetch().

View.**Modify**(kind, data)

Modify the view, by calling MsiViewModify(). *kind* can be one of MSIMODIFY_SEEK, MSIMODIFY_REFRESH, MSIMODIFY_INSERT, MSIMODIFY_UPDATE, MSIMODIFY_ASSIGN, MSIMODIFY_REPLACE, MSIMODIFY_MERGE, MSIMODIFY_INSERT_TEMPORARY, MSIMODIFY_VALIDATE, MSIMODIFY_VALIDATE_NEW, MSIMODIFY_VALIDATE_FIELD, or MSIMODIFY_VALIDATE_DELETE.

*data* must be a record describing the new data.

View.**Close**()

Close the view, through MsiViewClose().

See also: MSIViewExecute MsiViewGetColumnInfo MsiViewFetch MsiViewModify MsiViewClose
35.1.3. Summary Information Objects

**SummaryInformation.GetProperty(field)**

Return a property of the summary, through `MsiSummaryInfoGetProperty()`. *field* is the name of the property, and can be one of the constants `PID_CODEPAGE`, `PID_TITLE`, `PID_SUBJECT`, `PID_AUTHOR`, `PID_KEYWORDS`, `PID_COMMENTS`, `PID_TEMPLATE`, `PID_LASTAUTHOR`, `PID_REVNUMBER`, `PID_LASTPRINTED`, `PID_CREATE_DTM`, `PID_LASTSAVE_DTM`, `PID_PAGECOUNT`, `PID_WORDCOUNT`, `PID_CHARCOUNT`, `PID_APPNAME`, or `PID_SECURITY`.

**SummaryInformation.GetPropertyCount()**

Return the number of summary properties, through `MsiSummaryInfoGetPropertyCount()`.

**SummaryInformation.SetProperty(field, value)**

Set a property through `MsiSummaryInfoSetProperty()`. *field* can have the same values as in `GetProperty()`, *value* is the new value of the property. Possible value types are integer and string.

**SummaryInformation.Persist()**

Write the modified properties to the summary information stream, using `MsiSummaryInfoPersist()`.

See also: MsiSummaryInfoGetProperty
MsiSummaryInfoGetPropertyCount MsiSummaryInfoSetProperty
MsiSummaryInfoPersist
35.1.4. Record Objects

**Record. `GetFieldCount()`**
Return the number of fields of the record, through `MsiRecordGetFieldCount()`.

**Record. `GetInteger(field)`**
Return the value of `field` as an integer where possible. `field` must be an integer.

**Record. `GetString(field)`**
Return the value of `field` as a string where possible. `field` must be an integer.

**Record. `SetString(field, value)`**
Set `field` to `value` through `MsiRecordSetString()`. `field` must be an integer; `value` a string.

**Record. `SetStream(field, value)`**
Set `field` to the contents of the file named `value`, through `MsiRecordSetStream()`. `field` must be an integer; `value` a string.

**Record. `SetInteger(field, value)`**
Set `field` to `value` through `MsiRecordSetInteger()`. Both `field` and `value` must be an integer.

**Record. `ClearData()`**
Set all fields of the record to 0, through `MsiRecordClearData()`.

**See also:** `MsiRecordGetFieldCount` `MsiRecordSetString` `MsiRecordSetStream` `MsiRecordSetInteger` `MsiRecordClear`
35.1.5. Errors

All wrappers around MSI functions raise `MsiError`; the string inside the exception will contain more detail.
35.1.6. CAB Objects

*class msilib.CAB(name)*

The class CAB represents a CAB file. During MSI construction, files will be added simultaneously to the Files table, and to a CAB file. Then, when all files have been added, the CAB file can be written, then added to the MSI file.

`name` is the name of the CAB file in the MSI file.

*append(full, file, logical)*

Add the file with the pathname `full` to the CAB file, under the name `logical`. If there is already a file named `logical`, a new file name is created.

Return the index of the file in the CAB file, and the new name of the file inside the CAB file.

*commit(database)*

Generate a CAB file, add it as a stream to the MSI file, put it into the Media table, and remove the generated file from the disk.
35.1.7. Directory Objects

class msilib.Directory(database, cab, basedir, physical, logical, default[, componentflags])

Create a new directory in the Directory table. There is a current component at each point in time for the directory, which is either explicitly created through start_component(), or implicitly when files are added for the first time. Files are added into the current component, and into the cab file. To create a directory, a base directory object needs to be specified (can be None), the path to the physical directory, and a logical directory name. default specifies the DefaultDir slot in the directory table. componentflags specifies the default flags that new components get.

start_component([component[, feature[, flags[, keyfile[, uuid]]]]])

Add an entry to the Component table, and make this component the current component for this directory. If no component name is given, the directory name is used. If no feature is given, the current feature is used. If no flags are given, the directory's default flags are used. If no keyfile is given, the KeyPath is left null in the Component table.

add_file(file[, src[, version[, language]]])

Add a file to the current component of the directory, starting a new one if there is no current component. By default, the file name in the source and the file table will be identical. If the src file is specified, it is interpreted relative to the current directory. Optionally, a version and a language can be specified for the entry in the File table.

glob(pattern[, exclude])
Add a list of files to the current component as specified in the glob pattern. Individual files can be excluded in the *exclude* list.

```python
remove_pyc()
```

Remove `.pyc`/.`pyo` files on uninstall.

**See also:** Directory Table File Table Component Table FeatureComponents Table
35.1.8. Features

```python
class msilib.Feature(database, id, title, desc, display[, level=1[, parent[, directory[, attributes=0]]]])
```

Add a new record to the Feature table, using the values `id`, `parent.id`, `title`, `desc`, `display`, `level`, `directory`, and `attributes`. The resulting feature object can be passed to the `start_component()` method of `Directory`.

**set_current()**

Make this feature the current feature of `msilib`. New components are automatically added to the default feature, unless a feature is explicitly specified.

**See also:** Feature Table
35.1.9. GUI classes

`msilib` provides several classes that wrap the GUI tables in an MSI database. However, no standard user interface is provided; use `bdist_msi` to create MSI files with a user-interface for installing Python packages.

```python
class msilib.Control(dlg, name)
    Base class of the dialog controls. `dlg` is the dialog object the control belongs to, and `name` is the control’s name.

event(event, argument[, condition=1[, ordering]])
    Make an entry into the ControlEvent table for this control.

mapping(event, attribute)
    Make an entry into the EventMapping table for this control.

condition(action, condition)
    Make an entry into the ControlCondition table for this control.
```

```python
class msilib.RadioButtonGroup(dlg, name, property)
    Create a radio button control named `name`. `property` is the installer property that gets set when a radio button is selected.

    add(name, x, y, width, height, text[, value])
        Add a radio button named `name` to the group, at the coordinates `x`, `y`, `width`, `height`, and with the label `text`. If `value` is omitted, it defaults to `name`.
```

```python
class msilib.Dialog(db, name, x, y, w, h, attr, title, first, default, cancel)
    Return a new Dialog object. An entry in the Dialog table is made, with the specified coordinates, dialog attributes, title, name of the
first, default, and cancel controls.

**control**(name, type, x, y, width, height, attributes, property, text, control_next, help)

Return a new Control object. An entry in the Control table is made with the specified parameters.

This is a generic method; for specific types, specialized methods are provided.

**text**(name, x, y, width, height, attributes, text)

Add and return a Text control.

**bitmap**(name, x, y, width, height, text)

Add and return a Bitmap control.

**line**(name, x, y, width, height)

Add and return a Line control.

**pushbutton**(name, x, y, width, height, attributes, text, next_control)

Add and return a PushButton control.

**radiogroup**(name, x, y, width, height, attributes, property, text, next_control)

Add and return a RadioButtonGroup control.

**checkbox**(name, x, y, width, height, attributes, property, text, next_control)

Add and return a CheckBox control.

See also: Dialog Table Control Table Control Types ControlCondition Table ControlEvent Table EventMapping Table RadioButton Table
35.1.10. Precomputed tables

`msilib` provides a few subpackages that contain only schema and table definitions. Currently, these definitions are based on MSI version 2.0.

`msilib.schema`
This is the standard MSI schema for MSI 2.0, with the `tables` variable providing a list of table definitions, and `Validation_records` providing the data for MSI validation.

`msilib.sequence`
This module contains table contents for the standard sequence tables: `AdminExecuteSequence`, `AdminUISequence`, `AdvtExecuteSequence`, `InstallExecuteSequence`, and `InstallUISequence`.

`msilib.text`
This module contains definitions for the `UIText` and `ActionText` tables, for the standard installer actions.
35.2. msvcrt – Useful routines from the MS VC++ runtime

These functions provide access to some useful capabilities on Windows platforms. Some higher-level modules use these functions to build the Windows implementations of their services. For example, the getpass module uses this in the implementation of the getpass() function.

Further documentation on these functions can be found in the Platform API documentation.

The module implements both the normal and wide char variants of the console I/O api. The normal API deals only with ASCII characters and is of limited use for internationalized applications. The wide char API should be used where ever possible.
35.2.1. File Operations

`msvcrt.locking(fd, mode, nbytes)`
Lock part of a file based on file descriptor `fd` from the C runtime. Raises `IOError` on failure. The locked region of the file extends from the current file position for `nbytes` bytes, and may continue beyond the end of the file. `mode` must be one of the `LK_*` constants listed below. Multiple regions in a file may be locked at the same time, but may not overlap. Adjacent regions are not merged; they must be unlocked individually.

`msvcrt.LK_LOCK`
`msvcrt.LK_RLCK`
Locks the specified bytes. If the bytes cannot be locked, the program immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, `IOError` is raised.

`msvcrt.LK_NBLCK`
`msvcrt.LK_NBRLCK`
Locks the specified bytes. If the bytes cannot be locked, `IOError` is raised.

`msvcrt.LK_UNLCK`
Unlocks the specified bytes, which must have been previously locked.

`msvcrt.setmode(fd, flags)`
Set the line-end translation mode for the file descriptor `fd`. To set it to text mode, `flags` should be `os.O_TEXT`; for binary, it should be `os.O_BINARY`.

`msvcrt.open_osfhandle(handle, flags)`
Create a C runtime file descriptor from the file handle `handle`. The `flags` parameter should be a bitwise OR of `os.O_APPEND`,
os.O_RDONLY, and os.O_TEXT. The returned file descriptor may be used as a parameter to os.fdopen() to create a file object.

msvcrt.get_osfhandle(fd)
Return the file handle for the file descriptor fd. Raises IOError if fd is not recognized.
35.2.2. Console I/O

**msvcrt.**kbhit**()**
Return true if a keypress is waiting to be read.

**msvcrt.**getch**()**
Read a keypress and return the resulting character. Nothing is echoed to the console. This call will block if a keypress is not already available, but will not wait for Enter to be pressed. If the pressed key was a special function key, this will return '\\000' or '\xe0'; the next call will return the keycode. The Control-C keypress cannot be read with this function.

**msvcrt.**getwch**()**
Wide char variant of **getch()**, returning a Unicode value.

*New in version 2.6.*

**msvcrt.**getche**()**
Similar to **getch()**, but the keypress will be echoed if it represents a printable character.

**msvcrt.**getwche**()**
Wide char variant of **getche()**, returning a Unicode value.

*New in version 2.6.*

**msvcrt.**putch**(char)**
Print the character **char** to the console without buffering.

**msvcrt.**putwch**(unicode_char)**
Wide char variant of **putch()**, accepting a Unicode value.

*New in version 2.6.*
**msvcrt.** `ungetch(char)`

Cause the character `char` to be “pushed back” into the console buffer; it will be the next character read by `getch()` or `getche()`.

**msvcrt.** `ungetwch(unicode_char)`

Wide char variant of `ungetch()`, accepting a Unicode value.

*New in version 2.6.*
35.2.3. Other Functions

`msvcrt.heapmin()`

Force the `malloc()` heap to clean itself up and return unused blocks to the operating system. On failure, this raises `IOError`.
Note: The _winreg module has been renamed to winreg in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

New in version 2.0.

These functions expose the Windows registry API to Python. Instead of using an integer as the registry handle, a handle object is used to ensure that the handles are closed correctly, even if the programmer neglects to explicitly close them.

This module offers the following functions:

_winreg.CloseKey(hkey)
Closes a previously opened registry key. The hkey argument specifies a previously opened key.

Note: If hkey is not closed using this method (or via hkey.Close()), it is closed when the hkey object is destroyed by Python.

_winreg.ConnectRegistry(computer_name, key)
Establishes a connection to a predefined registry handle on another computer, and returns a handle object.

computer_name is the name of the remote computer, of the form r"\computernamename". If None, the local computer is used.

key is the predefined handle to connect to.
The return value is the handle of the opened key. If the function fails, a `WindowsError` exception is raised.

```python
_winreg.CreateKey(key, sub_key)
```

Creates or opens the specified key, returning a handle object.

`key` is an already open key, or one of the predefined `HKEY_*` constants.

`sub_key` is a string that names the key this method opens or creates.

If `key` is one of the predefined keys, `sub_key` may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, a `WindowsError` exception is raised.

```python
_winreg.CreateKeyEx(key, sub_key[, res[, sam]])
```

Creates or opens the specified key, returning a handle object.

`key` is an already open key, or one of the predefined `HKEY_*` constants.

`sub_key` is a string that names the key this method opens or creates.

`res` is a reserved integer, and must be zero. The default is zero.

`sam` is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_ALL_ACCESS`. See Access Rights for other allowed values.
If `key` is one of the predefined keys, `sub_key` may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, a `WindowsError` exception is raised.

*New in version 2.7.*

```python
_winreg.DeleteKey(key, sub_key)
```

Deletes the specified key.

- `key` is an already open key, or any one of the predefined `HKEY_*` constants.

- `sub_key` is a string that must be a subkey of the key identified by the `key` parameter. This value must not be `None`, and the key may not have subkeys.

*This method can not delete keys with subkeys.*

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, a `WindowsError` exception is raised.

```python
_winreg.DeleteKeyEx(key, sub_key[, sam[, res]])
```

Deletes the specified key.

**Note:** The `DeleteKeyEx()` function is implemented with the `RegDeleteKeyEx` Windows API function, which is specific to 64-bit versions of Windows. See the `RegDeleteKeyEx documentation`. 
key is an already open key, or any one of the predefined HKEY_*
constants.

sub_key is a string that must be a subkey of the key identified by
the key parameter. This value must not be None, and the key may
not have subkeys.

res is a reserved integer, and must be zero. The default is zero.

sam is an integer that specifies an access mask that describes
the desired security access for the key. Default is
KEY_WOW64_64KEY. See Access Rights for other allowed values.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values,
is removed. If the method fails, a WindowsError exception is
raised.

On unsupported Windows versions, NotImplemented Exception is
raised.

New in version 2.7.

_winreg. DeleteValue(key, value)
Removes a named value from a registry key.

key is an already open key, or one of the predefined HKEY_*
constants.

value is a string that identifies the value to remove.

_winreg. EnumKey(key, index)
Enumerates subkeys of an open registry key, returning a string.

key is an already open key, or any one of the predefined HKEY_*
 CONSTANTS

_\textit{index}_ is an integer that identifies the index of the key to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly until a \texttt{WindowsError} exception is raised, indicating, no more values are available.

\texttt{_winreg.EnumValue(key, index)}

Enumerates values of an open registry key, returning a tuple.

\texttt{key} is an already open key, or any one of the predefined \texttt{HKEY_*} constants.

\texttt{index} is an integer that identifies the index of the value to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly, until a \texttt{WindowsError} exception is raised, indicating no more values.

The result is a tuple of 3 items:

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A string that identifies the value name</td>
</tr>
<tr>
<td>1</td>
<td>An object that holds the value data, and whose type depends on the underlying registry type</td>
</tr>
<tr>
<td>2</td>
<td>An integer that identifies the type of the value data (see table in docs for \texttt{SetValueEx()})</td>
</tr>
</tbody>
</table>

\texttt{_winreg.ExpandEnvironmentStrings(unicode)}

Expands environment variable placeholders \texttt{%NAME%} in unicode strings like \texttt{REG_EXPAND_SZ}:

```python
>>> ExpandEnvironmentStrings(u"%windir%")
u"C:\Windows"
```
New in version 2.6.

_winreg. **FlushKey**(key)

Writes all the attributes of a key to the registry.

*key* is an already open key, or one of the predefined **HKEY_* constants**.

It is not necessary to call **FlushKey()** to change a key. Registry changes are flushed to disk by the registry using its lazy flusher. Registry changes are also flushed to disk at system shutdown. Unlike **CloseKey()**, the **FlushKey()** method returns only when all the data has been written to the registry. An application should only call **FlushKey()** if it requires absolute certainty that registry changes are on disk.

**Note:** If you don’t know whether a **FlushKey()** call is required, it probably isn’t.

_winreg. **LoadKey**(key, sub_key, file_name)

Creates a subkey under the specified key and stores registration information from a specified file into that subkey.

*key* is a handle returned by **ConnectRegistry()** or one of the constants **HKEY_USERS** or **HKEY_LOCAL_MACHINE**.

*sub_key* is a string that identifies the subkey to load.

*file_name* is the name of the file to load registry data from. This file must have been created with the **SaveKey()** function. Under the file allocation table (FAT) file system, the filename may not have an extension.

A call to **LoadKey()** fails if the calling process does not have the
SE_RESTORE_PRIVILEGE privilege. Note that privileges are different from permissions – see the RegLoadKey documentation for more details.

If key is a handle returned by ConnectRegistry(), then the path specified in file_name is relative to the remote computer.

_winreg.OpenKey(key, sub_key[], res[], sam[])
Opens the specified key, returning a handle object.

key is an already open key, or any one of the predefined HKEY_* constants.

sub_key is a string that identifies the sub_key to open.

res is a reserved integer, and must be zero. The default is zero.

sam is an integer that specifies an access mask that describes the desired security access for the key. Default is KEY_READ. See Access Rights for other allowed values.

The result is a new handle to the specified key.

If the function fails, WindowsError is raised.

_winreg.OpenKeyEx()

The functionality of OpenKeyEx() is provided via OpenKey(), by the use of default arguments.

_winreg.QueryInfoKey(key)
Returns information about a key, as a tuple.

key is an already open key, or one of the predefined HKEY_* constants.

The result is a tuple of 3 items:
Index | Meaning
---|---
0 | An integer giving the number of sub keys this key has.
1 | An integer giving the number of values this key has.
2 | A long integer giving when the key was last modified (if available) as 100’s of nanoseconds since Jan 1, 1601.

_winreg.**QueryValue**(key, sub_key)
Retrieves the unnamed value for a key, as a string.

*key* is an already open key, or one of the predefined **HKEY_* constants**.

*sub_key* is a string that holds the name of the subkey with which the value is associated. If this parameter is **None** or empty, the function retrieves the value set by the **SetValue()** method for the key identified by *key*.

Values in the registry have name, type, and data components. This method retrieves the data for a key's first value that has a NULL name. But the underlying API call doesn’t return the type, so always use **QueryValueEx()** if possible.

_winreg.**QueryValueEx**(key, value_name)
Retrieves the type and data for a specified value name associated with an open registry key.

*key* is an already open key, or one of the predefined **HKEY_* constants**.

*value_name* is a string indicating the value to query.

The result is a tuple of 2 items:

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The value of the registry item.</td>
</tr>
</tbody>
</table>
_winreg. **SaveKey(key, file_name)**

Saves the specified key, and all its subkeys to the specified file.

*key* is an already open key, or one of the predefined HKEY_* constants.

*file_name* is the name of the file to save registry data to. This file cannot already exist. If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the `LoadKey()` method.

If *key* represents a key on a remote computer, the path described by *file_name* is relative to the remote computer. The caller of this method must possess the SeBackupPrivilege security privilege. Note that privileges are different than permissions – see the Conflicts Between User Rights and Permissions documentation for more details.

This function passes NULL for security_attributes to the API.

_\_winreg. **SetValue(key, sub_key, type, value)**

Associates a value with a specified key.

*key* is an already open key, or one of the predefined HKEY_* constants.

*sub_key* is a string that names the subkey with which the value is associated.

*type* is an integer that specifies the type of the data. Currently this must be REG_SZ, meaning only strings are supported. Use the `SetValueEx()` function for support for other data types.
value is a string that specifies the new value.

If the key specified by the sub_key parameter does not exist, the SetValue function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

The key identified by the key parameter must have been opened with KEY_SET_VALUE access.

_WINREG.SetValueEx(key, value_name, reserved, type, value)
Stores data in the value field of an open registry key.

key is an already open key, or one of the predefined HKEY_*
constants.

value_name is a string that names the subkey with which the value is associated.

type is an integer that specifies the type of the data. See Value Types for the available types.

reserved can be anything – zero is always passed to the API.

value is a string that specifies the new value.

This method can also set additional value and type information for the specified key. The key identified by the key parameter must have been opened with KEY_SET_VALUE access.

To open the key, use the CreateKey() or OpenKey() methods.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the
filenames stored in the configuration registry. This helps the registry perform efficiently.

_winreg. **DisableReflectionKey(key)**
Disables registry reflection for 32-bit processes running on a 64-bit operating system.

*key* is an already open key, or one of the predefined HKEY_* constants.

Will generally raise **NotImplemented** if executed on a 32-bit operating system.

If the key is not on the reflection list, the function succeeds but has no effect. Disabling reflection for a key does not affect reflection of any subkeys.

_winreg. **EnableReflectionKey(key)**
Restores registry reflection for the specified disabled key.

*key* is an already open key, or one of the predefined HKEY_* constants.

Will generally raise **NotImplemented** if executed on a 32-bit operating system.

Restoring reflection for a key does not affect reflection of any subkeys.

_winreg. **QueryReflectionKey(key)**
Determines the reflection state for the specified key.

*key* is an already open key, or one of the predefined HKEY_* constants.

Returns True if reflection is disabled.
Will generally raise `NotImplemented` if executed on a 32-bit operating system.
35.3.1. Constants

The following constants are defined for use in many _winreg
functions.

35.3.1.1. HKEY_* Constants

_winreg. HKEY_CLASSES_ROOT
Registry entries subordinate to this key define types (or classes) of
documents and the properties associated with those types. Shell and COM applications use the information stored under this key.

_winreg. HKEY_CURRENT_USER
Registry entries subordinate to this key define the preferences of the current user. These preferences include the settings of environment variables, data about program groups, colors, printers, network connections, and application preferences.

_winreg. HKEY_LOCAL_MACHINE
Registry entries subordinate to this key define the physical state of the computer, including data about the bus type, system memory, and installed hardware and software.

_winreg. HKEY_USERS
Registry entries subordinate to this key define the default user configuration for new users on the local computer and the user configuration for the current user.

_winreg. HKEY_PERFORMANCE_DATA
Registry entries subordinate to this key allow you to access performance data. The data is not actually stored in the registry; the registry functions cause the system to collect the data from its source.
_winreg.HKEY_CURRENT_CONFIG
Contains information about the current hardware profile of the local computer system.

_winreg.HKEY_DYN_DATA
This key is not used in versions of Windows after 98.

35.3.1.2. Access Rights

For more information, see Registry Key Security and Access.

_winreg.KEY_ALL_ACCESS
Combines the STANDARD_RIGHTS_REQUIRED, KEY_QUERY_VALUE, KEY_SET_VALUE, KEY_CREATE_SUB_KEY,
KEY_ENUMERATE_SUB_KEYS, KEY_NOTIFY, and KEY_CREATE_LINK access rights.

_winreg.KEY_WRITE
Combines the STANDARD_RIGHTS_WRITE, KEY_SET_VALUE, and KEY_CREATE_SUB_KEY access rights.

_winreg.KEY_READ
Combines the STANDARD_RIGHTS_READ, KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, and KEY_NOTIFY values.

_winreg.KEY_EXECUTE
Equivalent to KEY_READ.

_winreg.KEY_QUERY_VALUE
Required to query the values of a registry key.

_winreg.KEY_SET_VALUE
Required to create, delete, or set a registry value.

_winreg.KEY_CREATE_SUB_KEY
Required to create a subkey of a registry key.
__winreg__\_.KEY\_ENUMERATE\_SUB\_KEYS
Required to enumerate the subkeys of a registry key.

__winreg__\_.KEY\_NOTIFY
Required to request change notifications for a registry key or for subkeys of a registry key.

__winreg__\_.KEY\_CREATE\_LINK
Reserved for system use.

### 35.3.1.2.1. 64-bit Specific

For more information, see Accessing an Alternate Registry View.

__winreg__\_.KEY\_WOW64\_64KEY
Indicates that an application on 64-bit Windows should operate on the 64-bit registry view.

__winreg__\_.KEY\_WOW64\_32KEY
Indicates that an application on 64-bit Windows should operate on the 32-bit registry view.

### 35.3.1.3. Value Types

For more information, see Registry Value Types.

__winreg__\_.REG\_BINARY
Binary data in any form.

__winreg__\_.REG\_DWORD
32-bit number.

__winreg__\_.REG\_DWORD\_LITTLE\_ENDIAN
A 32-bit number in little-endian format.

__winreg__\_.REG\_DWORD\_BIG\_ENDIAN
A 32-bit number in big-endian format.
_winreg.REG_EXPAND_SZ
   Null-terminated string containing references to environment variables (%PATH%).

_winreg.REG_LINK
   A Unicode symbolic link.

_winreg.REG_MULTI_SZ
   A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination automatically.)

_winreg.REG_NONE
   No defined value type.

_winreg.REG_RESOURCE_LIST
   A device-driver resource list.

_winreg.REG_FULL_RESOURCE_DESCRIPTOR
   A hardware setting.

_winreg.REG_RESOURCE_REQUIREMENTS_LIST
   A hardware resource list.

_winreg.REG_SZ
   A null-terminated string.
35.3.2. Registry Handle Objects

This object wraps a Windows HKEY object, automatically closing it when the object is destroyed. To guarantee cleanup, you can call either the `Close()` method on the object, or the `CloseKey()` function.

All registry functions in this module return one of these objects.

All registry functions in this module which accept a handle object also accept an integer, however, use of the handle object is encouraged.

Handle objects provide semantics for `__nonzero__()` – thus:

```python
if handle:
    print "Yes"
```

will print `Yes` if the handle is currently valid (has not been closed or detached).

The object also support comparison semantics, so handle objects will compare true if they both reference the same underlying Windows handle value.

Handle objects can be converted to an integer (e.g., using the built-in `int()` function), in which case the underlying Windows handle value is returned. You can also use the `Detach()` method to return the integer handle, and also disconnect the Windows handle from the handle object.

```python
PyHKEY.Close()
```

Closes the underlying Windows handle.

If the handle is already closed, no error is raised.
PyHKEY. **Detach()**

Detaches the Windows handle from the handle object.

The result is an integer (or long on 64 bit Windows) that holds the value of the handle before it is detached. If the handle is already detached or closed, this will return zero.

After calling this function, the handle is effectively invalidated, but the handle is not closed. You would call this function when you need the underlying Win32 handle to exist beyond the lifetime of the handle object.

PyHKEY. **__enter__()**

PyHKEY. **__exit__(*exc_info)**

The HKEY object implements **__enter__()** and **__exit__()** and thus supports the context protocol for the **with** statement:

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ...
    # work with key
```

will automatically close `key` when control leaves the **with** block.

*New in version 2.6.*
35.4. winsound — Sound-playing interface for Windows

New in version 1.5.2.

The winsound module provides access to the basic sound-playing machinery provided by Windows platforms. It includes functions and several constants.

winsound.**Beep**(frequency, duration)

Beep the PC’s speaker. The *frequency* parameter specifies frequency, in hertz, of the sound, and must be in the range 37 through 32,767. The *duration* parameter specifies the number of milliseconds the sound should last. If the system is not able to beep the speaker, *RuntimeError* is raised.

New in version 1.6.

winsound.**PlaySound**(sound, flags)

Call the underlying PlaySound() function from the Platform API. The *sound* parameter may be a filename, audio data as a string, or *None*. Its interpretation depends on the value of *flags*, which can be a bitwise ORed combination of the constants described below. If the *sound* parameter is *None*, any currently playing waveform sound is stopped. If the system indicates an error, *RuntimeError* is raised.

winsound.**MessageBeep**(type=MB_OK)

Call the underlying MessageBeep() function from the Platform API. This plays a sound as specified in the registry. The *type* argument specifies which sound to play; possible values are -1, MB_ICONASTERISK, MB_ICONEXCLAMATION, MB_ICONHAND,
MB_ICONQUESTION, and MB_OK, all described below. The value -1 produces a “simple beep”; this is the final fallback if a sound cannot be played otherwise.

New in version 2.3.

winsound. SND_FILENAME
The sound parameter is the name of a WAV file. Do not use with SND_ALIAS.

winsound. SND_ALIAS
The sound parameter is a sound association name from the registry. If the registry contains no such name, play the system default sound unless SND_NODEFAULT is also specified. If no default sound is registered, raise RuntimeError. Do not use with SND_FILENAME.

All Win32 systems support at least the following; most systems support many more:

<table>
<thead>
<tr>
<th>PlaySound() name</th>
<th>Corresponding Control Panel Sound name</th>
</tr>
</thead>
<tbody>
<tr>
<td>'SystemAsterisk'</td>
<td>Asterisk</td>
</tr>
<tr>
<td>'SystemExclamation'</td>
<td>Exclamation</td>
</tr>
<tr>
<td>'SystemExit'</td>
<td>Exit Windows</td>
</tr>
<tr>
<td>'SystemHand'</td>
<td>Critical Stop</td>
</tr>
<tr>
<td>'SystemQuestion'</td>
<td>Question</td>
</tr>
</tbody>
</table>

For example:

```python
import winsound
#
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered
# "*" probably isn't the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```
**winsound. SND_LOOP**

Play the sound repeatedly. The SND_ASYNC flag must also be used to avoid blocking. Cannot be used with SND_MEMORY.

**winsound. SND_MEMORY**

The sound parameter to `PlaySound()` is a memory image of a WAV file, as a string.

**Note:** This module does not support playing from a memory image asynchronously, so a combination of this flag and SND_ASYNC will raise RuntimeError.

**winsound. SND_PURGE**

Stop playing all instances of the specified sound.

**Note:** This flag is not supported on modern Windows platforms.

**winsound. SND_ASYNC**

Return immediately, allowing sounds to play asynchronously.

**winsound. SND_NODEFAULT**

If the specified sound cannot be found, do not play the system default sound.

**winsound. SND_NOSTOP**

Do not interrupt sounds currently playing.

**winsound. SND_NOWAIT**

Return immediately if the sound driver is busy.

**Note:** This flag is not supported on modern Windows platforms.
winsound.**MB_ICONASTERISK**

Play the `SystemDefault` sound.

winsound.**MB_ICONEXCLAMATION**

Play the `SystemExclamation` sound.

winsound.**MB_ICONHAND**

Play the `SystemHand` sound.

winsound.**MB_ICONQUESTION**

Play the `SystemQuestion` sound.

winsound.**MB_OK**

Play the `SystemDefault` sound.
36. Unix Specific Services

The modules described in this chapter provide interfaces to features that are unique to the Unix operating system, or in some cases to some or many variants of it. Here's an overview:

- 36.1. posix — The most common POSIX system calls
  - 36.1.1. Large File Support
  - 36.1.2. Notable Module Contents
- 36.2. pwd — The password database
- 36.3. spwd — The shadow password database
- 36.4. grp — The group database
- 36.5. crypt — Function to check Unix passwords
- 36.6. dl — Call C functions in shared objects
  - 36.6.1. DL Objects
- 36.7. termios — POSIX style tty control
  - 36.7.1. Example
- 36.8. tty — Terminal control functions
- 36.9. pty — Pseudo-terminal utilities
- 36.10. fcntl — The fcntl and ioctl system calls
- 36.11. pipes — Interface to shell pipelines
  - 36.11.1. Template Objects
- 36.12. posixfile — File-like objects with locking support
- 36.13. resource — Resource usage information
  - 36.13.1. Resource Limits
  - 36.13.2. Resource Usage
- 36.14. nis — Interface to Sun’s NIS (Yellow Pages)
- 36.15. syslog — Unix syslog library routines
  - 36.15.1. Examples
    - 36.15.1.1. Simple example
- 36.16. commands — Utilities for running commands
This module provides access to operating system functionality that is standardized by the C Standard and the POSIX standard (a thinly disguised Unix interface).

**Do not import this module directly.** Instead, import the module *os*, which provides a *portable* version of this interface. On Unix, the *os* module provides a superset of the *posix* interface. On non-Unix operating systems the *posix* module is not available, but a subset is always available through the *os* interface. Once *os* is imported, there is *no* performance penalty in using it instead of *posix*. In addition, *os* provides some additional functionality, such as automatically calling `putenv()` when an entry in *os.environ* is changed.

Errors are reported as exceptions; the usual exceptions are given for type errors, while errors reported by the system calls raise *OSError*. 

---

36.1. **posix** — The most common POSIX system calls
36.1.1. Large File Support

Several operating systems (including AIX, HP-UX, Irix and Solaris) provide support for files that are larger than 2 GB from a C programming model where `int` and `long` are 32-bit values. This is typically accomplished by defining the relevant size and offset types as 64-bit values. Such files are sometimes referred to as *large files*.

Large file support is enabled in Python when the size of an `off_t` is larger than a `long` and the `long long` type is available and is at least as large as an `off_t`. Python longs are then used to represent file sizes, offsets and other values that can exceed the range of a Python `int`. It may be necessary to configure and compile Python with certain compiler flags to enable this mode. For example, it is enabled by default with recent versions of Irix, but with Solaris 2.6 and 2.7 you need to do something like:

```
CFLAGS="`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \
./configure
```

On large-file-capable Linux systems, this might work:

```
CFLAGS='"-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64"' OPT="-g -O2 $CFLAGS" \
./configure
```
36.1.2. Notable Module Contents

In addition to many functions described in the `os` module documentation, `posix` defines the following data item:

```python
posix.environ
```

A dictionary representing the string environment at the time the interpreter was started. For example, `environ['HOME']` is the pathname of your home directory, equivalent to `getenv("HOME")` in C.

Modifying this dictionary does not affect the string environment passed on by `execv()`, `popen()` or `system()`; if you need to change the environment, pass `environ` to `execve()` or add variable assignments and export statements to the command string for `system()` or `popen()`.

**Note:** The `os` module provides an alternate implementation of `environ` which updates the environment on modification. Note also that updating `os.environ` will render this dictionary obsolete. Use of the `os` module version of this is recommended over direct access to the `posix` module.
36.2. `pwd` — The password database

This module provides access to the Unix user account and password database. It is available on all Unix versions.

Password database entries are reported as a tuple-like object, whose attributes correspond to the members of the `passwd` structure (Attribute field below, see `<pwd.h>`):

<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>pw_name</code></td>
<td>Login name</td>
</tr>
<tr>
<td>1</td>
<td><code>pw_passwd</code></td>
<td>Optional encrypted password</td>
</tr>
<tr>
<td>2</td>
<td><code>pw_uid</code></td>
<td>Numerical user ID</td>
</tr>
<tr>
<td>3</td>
<td><code>pw_gid</code></td>
<td>Numerical group ID</td>
</tr>
<tr>
<td>4</td>
<td><code>pw_gecos</code></td>
<td>User name or comment field</td>
</tr>
<tr>
<td>5</td>
<td><code>pw_dir</code></td>
<td>User home directory</td>
</tr>
<tr>
<td>6</td>
<td><code>pw_shell</code></td>
<td>User command interpreter</td>
</tr>
</tbody>
</table>

The uid and gid items are integers, all others are strings. `KeyError` is raised if the entry asked for cannot be found.

**Note:** In traditional Unix the field `pw_passwd` usually contains a password encrypted with a DES derived algorithm (see module `crypt`). However most modern unices use a so-called shadow password system. On those unices the `pw_passwd` field only contains an asterisk (`'*'`) or the letter `x` where the encrypted password is stored in a file `/etc/shadow` which is not world readable. Whether the `pw_passwd` field contains anything useful is system-dependent. If available, the `spwd` module should be used where access to the encrypted password is required.
It defines the following items:

```
pwd. getpwuid(uid)
    Return the password database entry for the given numeric user ID.
```

```
pwd. getpwnam(name)
    Return the password database entry for the given user name.
```

```
pwd. getpwall()
    Return a list of all available password database entries, in arbitrary order.
```

See also:

**Module** `grp`
   An interface to the group database, similar to this.

**Module** `spwd`
   An interface to the shadow password database, similar to this.
36.3. `spwd` — The shadow password database

*New in version 2.5.*

This module provides access to the Unix shadow password database. It is available on various Unix versions.

You must have enough privileges to access the shadow password database (this usually means you have to be root).

Shadow password database entries are reported as a tuple-like object, whose attributes correspond to the members of the `spwd` structure (Attribute field below, see `<shadow.h>`):

<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>sp_nam</code></td>
<td>Login name</td>
</tr>
<tr>
<td>1</td>
<td><code>sp_pwd</code></td>
<td>Encrypted password</td>
</tr>
<tr>
<td>2</td>
<td><code>sp_lstchg</code></td>
<td>Date of last change</td>
</tr>
<tr>
<td>3</td>
<td><code>sp_min</code></td>
<td>Minimal number of days between changes</td>
</tr>
<tr>
<td>4</td>
<td><code>sp_max</code></td>
<td>Maximum number of days between changes</td>
</tr>
<tr>
<td>5</td>
<td><code>sp_warn</code></td>
<td>Number of days before password expires to warn user about it</td>
</tr>
<tr>
<td>6</td>
<td><code>sp_inact</code></td>
<td>Number of days after password expires until account is blocked</td>
</tr>
<tr>
<td>7</td>
<td><code>sp_expire</code></td>
<td>Number of days since 1970-01-01 until account is disabled</td>
</tr>
<tr>
<td>8</td>
<td><code>sp_flag</code></td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The `sp_nam` and `sp_pwd` items are strings, all others are integers. `KeyError` is raised if the entry asked for cannot be found.
It defines the following items:

```
spwd.getspnam(name)
```
Return the shadow password database entry for the given user name.

```
spwd.getspall()
```
Return a list of all available shadow password database entries, in arbitrary order.

See also:

**Module grp**
An interface to the group database, similar to this.

**Module pwd**
An interface to the normal password database, similar to this.
This module provides access to the Unix group database. It is available on all Unix versions.

Group database entries are reported as a tuple-like object, whose attributes correspond to the members of the `group` structure (Attribute field below, see `<pwd.h>`):

<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>gr_name</td>
<td>the name of the group</td>
</tr>
<tr>
<td>1</td>
<td>gr_passwd</td>
<td>the (encrypted) group password; often empty</td>
</tr>
<tr>
<td>2</td>
<td>gr_gid</td>
<td>the numerical group ID</td>
</tr>
<tr>
<td>3</td>
<td>gr_mem</td>
<td>all the group member’s user names</td>
</tr>
</tbody>
</table>

The gid is an integer, name and password are strings, and the member list is a list of strings. (Note that most users are not explicitly listed as members of the group they are in according to the password database. Check both databases to get complete membership information. Also note that a `gr_name` that starts with a + or - is likely to be a YP/NIS reference and may not be accessible via `getgrnam()` or `getgrgid()`.)

It defines the following items:

- `grp.getgrgid(gid)`
  - Return the group database entry for the given numeric group ID. `KeyError` is raised if the entry asked for cannot be found.

- `grp.getgrnam(name)`
  - Return the group database entry for the given group name. `KeyError` is raised if the entry asked for cannot be found.
grp. `getgrall()`

Return a list of all available group entries, in arbitrary order.

**See also:**

**Module `pwd`**

An interface to the user database, similar to this.

**Module `spwd`**

An interface to the shadow password database, similar to this.
36.5. crypt — Function to check Unix passwords

This module implements an interface to the crypt(3) routine, which is a one-way hash function based upon a modified DES algorithm; see the Unix man page for further details. Possible uses include allowing Python scripts to accept typed passwords from the user, or attempting to crack Unix passwords with a dictionary.

Notice that the behavior of this module depends on the actual implementation of the crypt(3) routine in the running system. Therefore, any extensions available on the current implementation will also be available on this module.

```python
crypt.crypt(word, salt)
```

word will usually be a user’s password as typed at a prompt or in a graphical interface. salt is usually a random two-character string which will be used to perturb the DES algorithm in one of 4096 ways. The characters in salt must be in the set `[./a-zA-Z0-9]`. Returns the hashed password as a string, which will be composed of characters from the same alphabet as the salt (the first two characters represent the salt itself).

Since a few crypt(3) extensions allow different values, with different sizes in the salt, it is recommended to use the full crypted password as salt when checking for a password.

A simple example illustrating typical use:

```python
import crypt, getpass, pwd

def login():
    username = raw_input('Python login:')
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
```
if cryptedpasswd == 'x' or cryptedpasswd == '*':
    raise NotImplementedError("Sorry, currently no support for shadow password")
    cleartext = getpass.getpass()
    return crypt.crypt(cleartext, cryptedpasswd) == cryptedpasswd
else:
    return 1
36.6. **d1** — Call C functions in shared objects

*Deprecated since version 2.6:* The **d1** module has been removed in Python 3. Use the **ctypes** module instead.

The **d1** module defines an interface to the **dlopen()** function, which is the most common interface on Unix platforms for handling dynamically linked libraries. It allows the program to call arbitrary functions in such a library.

**Warning:** The **d1** module bypasses the Python type system and error handling. If used incorrectly it may cause segmentation faults, crashes or other incorrect behaviour.

**Note:** This module will not work unless **sizeof(int) == sizeof(long) == sizeof(char *)**. If this is not the case, **SystemError** will be raised on import.

The **d1** module defines the following function:

```python
d1.open(name[, mode=RTLD_LAZY])
```

Open a shared object file, and return a handle. Mode signifies late binding (**RTLD_LAZY**) or immediate binding (**RTLD_NOW**). Default is **RTLD_LAZY**. Note that some systems do not support **RTLD_NOW**.

Return value is a **d1object**.

The **d1** module defines the following constants:

```python
d1.RTLD_LAZY
```
Useful as an argument to `open()`.

**dl. RTLD_NOW**

Useful as an argument to `open()`. Note that on systems which do not support immediate binding, this constant will not appear in the module. For maximum portability, use `hasattr()` to determine if the system supports immediate binding.

The **dl** module defines the following exception:

*exception dl.error*

Exception raised when an error has occurred inside the dynamic loading and linking routines.

Example:

```python
>>> import dl, time
>>> a=dl.open('/lib/libc.so.6')
>>> a.call('time'), time.time()
(929723914, 929723914.498)
```

This example was tried on a Debian GNU/Linux system, and is a good example of the fact that using this module is usually a bad alternative.
36.6.1. DI Objects

DI objects, as returned by `open()` above, have the following methods:

**dl.close()**
Free all resources, except the memory.

**dl.sym(name)**
Return the pointer for the function named `name`, as a number, if it exists in the referenced shared object, otherwise `None`. This is useful in code like:

```python
>>> if a.sym('time'):
...     a.call('time')
... else:
...     time.time()
```

(Note that this function will return a non-zero number, as zero is the `NULL` pointer)

**dl.call(name[, arg1[, arg2...]])**
Call the function named `name` in the referenced shared object. The arguments must be either Python integers, which will be passed as is, Python strings, to which a pointer will be passed, or `None`, which will be passed as `NULL`. Note that strings should only be passed to functions as `const char*`, as Python will not like its string mutated.

There must be at most 10 arguments, and arguments not given will be treated as `None`. The function’s return value must be a C `long`, which is a Python integer.
This module provides an interface to the POSIX calls for tty I/O control. For a complete description of these calls, see termios(2) Unix manual page. It is only available for those Unix versions that support POSIX termios style tty I/O control configured during installation.

All functions in this module take a file descriptor \textit{fd} as their first argument. This can be an integer file descriptor, such as returned by \texttt{sys.stdin.fileno()}, or a file object, such as \texttt{sys.stdin} itself.

This module also defines all the constants needed to work with the functions provided here; these have the same name as their counterparts in C. Please refer to your system documentation for more information on using these terminal control interfaces.

The module defines the following functions:

\begin{description}
\item[termios.tcgetattr(fd)]
Return a list containing the tty attributes for file descriptor \textit{fd}, as follows: \texttt{[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]} where \textit{cc} is a list of the tty special characters (each a string of length 1, except the items with indices \texttt{VMIN} and \texttt{VTIME}, which are integers when these fields are defined). The interpretation of the flags and the speeds as well as the indexing in the \textit{cc} array must be done using the symbolic constants defined in the \texttt{termios} module.
\item[termios.tcsetattr(fd, when, attributes)]
Set the tty attributes for file descriptor \textit{fd} from the \textit{attributes}, which is a list like the one returned by \texttt{tcgetattr()}. The \textit{when}
argument determines when the attributes are changed: \texttt{TCSANOW} to change immediately, \texttt{TCSADRAIN} to change after transmitting all queued output, or \texttt{TCSAFLUSH} to change after transmitting all queued output and discarding all queued input.

\texttt{termios.tcsendbreak(fd, duration)}

Send a break on file descriptor \emph{fd}. A zero \emph{duration} sends a break for 0.25 –0.5 seconds; a nonzero \emph{duration} has a system dependent meaning.

\texttt{termios.tcdrain(fd)}

Wait until all output written to file descriptor \emph{fd} has been transmitted.

\texttt{termios.tcflush(fd, queue)}

Discard queued data on file descriptor \emph{fd}. The \emph{queue} selector specifies which queue: \texttt{TCIFLUSH} for the input queue, \texttt{TCOFLUSH} for the output queue, or \texttt{TCIOFLUSH} for both queues.

\texttt{termios.tcflow(fd, action)}

Suspend or resume input or output on file descriptor \emph{fd}. The \emph{action} argument can be \texttt{TCOOff} to suspend output, \texttt{TCOOn} to restart output, \texttt{TCIOFF} to suspend input, or \texttt{TCION} to restart input.

\texttt{See also:}

\texttt{Module tty}

Convenience functions for common terminal control operations.
36.7.1. Example

Here's a function that prompts for a password with echoing turned off. Note the technique using a separate `tcgetattr()` call and a `try ... finally` statement to ensure that the old tty attributes are restored exactly no matter what happens:

```python
def getpass(prompt=\"Password: \"):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = raw_input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

© Copyright 1990-2016, Python Software Foundation. Please donate.

Last updated on Dec 17, 2016.

Found a bug? Created using Sphinx 1.4.4.
The `tty` module defines functions for putting the tty into cbreak and raw modes.

Because it requires the `termios` module, it will work only on Unix.

The `tty` module defines the following functions:

- `tty.setraw(fd[, when])`
  Change the mode of the file descriptor `fd` to raw. If `when` is omitted, it defaults to `termios.TCSAFLUSH`, and is passed to `termios.tcsetattr()`.

- `tty.setcbreak(fd[, when])`
  Change the mode of file descriptor `fd` to cbreak. If `when` is omitted, it defaults to `termios.TCSAFLUSH`, and is passed to `termios.tcsetattr()`.

**See also:**

- **Module termios**
  Low-level terminal control interface.
The `pty` module defines operations for handling the pseudo-terminal concept: starting another process and being able to write to and read from its controlling terminal programatically.

Because pseudo-terminal handling is highly platform dependent, there is code to do it only for Linux. (The Linux code is supposed to work on other platforms, but hasn’t been tested yet.)

The `pty` module defines the following functions:

**pty. fork()**
Fork. Connect the child’s controlling terminal to a pseudo-terminal. Return value is `(pid, fd)`. Note that the child gets `pid 0`, and the `fd` is `invalid`. The parent’s return value is the `pid` of the child, and `fd` is a file descriptor connected to the child’s controlling terminal (and also to the child’s standard input and output).

**pty. openpty()**
Open a new pseudo-terminal pair, using `os.openpty()` if possible, or emulation code for generic Unix systems. Return a pair of file descriptors `(master, slave)`, for the master and the slave end, respectively.

**pty. spawn(argv[, master_read[, stdin_read]])**
Spawn a process, and connect its controlling terminal with the current process’s standard io. This is often used to baffle programs which insist on reading from the controlling terminal.

The functions `master_read` and `stdin_read` should be functions which read from a file descriptor. The defaults try to read 1024
bytes each time they are called.
This module performs file control and I/O control on file descriptors. It is an interface to the `fcntl()` and `ioctl()` Unix routines. For a complete description of these calls, see `fcntl(2)` and `ioctl(2)` Unix manual pages.

All functions in this module take a file descriptor *fd* as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or a file object, such as `sys.stdin` itself, which provides a `fileno()` which returns a genuine file descriptor.

The module defines the following functions:

```python
fcntl. fcntl(fd, op[, arg])
```

Perform the operation *op* on file descriptor *fd* (file objects providing a `fileno()` method are accepted as well). The values used for *op* are operating system dependent, and are available as constants in the `fcntl` module, using the same names as used in the relevant C header files. The argument *arg* is optional, and defaults to the integer value 0. When present, it can either be an integer value, or a string. With the argument missing or an integer value, the return value of this function is the integer return value of the C `fcntl()` call. When the argument is a string it represents a binary structure, e.g. created by `struct.pack()`. The binary data is copied to a buffer whose address is passed to the C `fcntl()` call. The return value after a successful call is the contents of the buffer, converted to a string object. The length of the returned string will be the same as the length of the *arg* argument. This is limited to 1024 bytes. If the information returned in the buffer by the operating system is
larger than 1024 bytes, this is most likely to result in a segmentation violation or a more subtle data corruption.

If the `fcntl()` fails, an `IOError` is raised.

```
fcntl.ioctl(fd, op[, arg[, mutate_flag]])
```

This function is identical to the `fcntl()` function, except that the operations are typically defined in the library module `termios` and the argument handling is even more complicated.

The `op` parameter is limited to values that can fit in 32-bits. Additional constants of interest for use as the `op` argument can be found in the `termios` module, under the same names as used in the relevant C header files.

The parameter `arg` can be one of an integer, absent (treated identically to the integer 0), an object supporting the read-only buffer interface (most likely a plain Python string) or an object supporting the read-write buffer interface.

In all but the last case, behaviour is as for the `fcntl()` function.

If a mutable buffer is passed, then the behaviour is determined by the value of the `mutate_flag` parameter.

If it is false, the buffer’s mutability is ignored and behaviour is as for a read-only buffer, except that the 1024 byte limit mentioned above is avoided – so long as the buffer you pass is as least as long as what the operating system wants to put there, things should work.

If `mutate_flag` is true, then the buffer is (in effect) passed to the underlying `ioctl()` system call, the latter’s return code is passed back to the calling Python, and the buffer’s new contents reflect the action of the `ioctl()`. This is a slight simplification, because if
the supplied buffer is less than 1024 bytes long it is first copied into a static buffer 1024 bytes long which is then passed to \texttt{ioctl()} and copied back into the supplied buffer.

If \textit{mutate\_flag} is not supplied, then from Python 2.5 it defaults to true, which is a change from versions 2.3 and 2.4. Supply the argument explicitly if version portability is a priority.

If the \texttt{ioctl()} fails, an \texttt{IOError} exception is raised.

An example:

\begin{verbatim}
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, "
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
\end{verbatim}

\texttt{fcntl.flock(fd, op)}

Perform the lock operation \textit{op} on file descriptor \textit{fd} (file objects providing a \texttt{fileno()} method are accepted as well). See the Unix manual \texttt{flock(2)} for details. (On some systems, this function is emulated using \texttt{fcntl()}.)

If the \texttt{flock()} fails, an \texttt{IOError} exception is raised.

\texttt{fcntl.lockf(fd, operation[, length[, start[, whence]]]])}

This is essentially a wrapper around the \texttt{fcntl()} locking calls. \textit{fd} is the file descriptor of the file to lock or unlock, and \textit{operation} is one of the following values:
- **LOCK_UN** – unlock
- **LOCK_SH** – acquire a shared lock
- **LOCK_EX** – acquire an exclusive lock

When *operation* is **LOCK_SH** or **LOCK_EX**, it can also be bitwise ORed with **LOCK_NB** to avoid blocking on lock acquisition. If **LOCK_NB** is used and the lock cannot be acquired, an **IOError** will be raised and the exception will have an *errno* attribute set to **EACCES** or **EAGAIN** (depending on the operating system; for portability, check for both values). On at least some systems, **LOCK_EX** can only be used if the file descriptor refers to a file opened for writing.

*length* is the number of bytes to lock, *start* is the byte offset at which the lock starts, relative to *whence*, and *whence* is as with *io.IOBase.seek()* specifically:

- 0 – relative to the start of the file (**os.SEEK_SET**)
- 1 – relative to the current buffer position (**os.SEEK_CUR**)
- 2 – relative to the end of the file (**os.SEEK_END**)

The default for *start* is 0, which means to start at the beginning of the file. The default for *length* is 0 which means to lock to the end of the file. The default for *whence* is also 0.

Examples (all on a SVR4 compliant system):

```python
import struct, fcntl, os

f = open(...)  # Open the file
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

Note that in the first example the return value variable *rv* will hold an integer value; in the second example it will hold a string value. The
structure lay-out for the lockdata variable is system dependent — therefore using the flock() call may be better.

See also:

Module os

If the locking flags O_SHLOCK and O_EXLOCK are present in the os module (on BSD only), the os.open() function provides an alternative to the lockf() and flock() functions.
36.11. pipes — Interface to shell pipelines

Source code: Lib/pipes.py

The pipes module defines a class to abstract the concept of a pipeline — a sequence of converters from one file to another.

Because the module uses /bin/sh command lines, a POSIX or compatible shell for os.system() and os.popen() is required.

class pipes.Template
    An abstraction of a pipeline.

Example:

```python
>>> import pipes
>>> t = pipes.Template()
>>> t.append('tr a-z A-Z', '---')
>>> f = t.open('pipefile', 'w')
>>> f.write('hello world')
>>> f.close()
>>> open('pipefile').read()
'HELLO WORLD'
```

pipes.quote(s)

Deprecated since version 2.7: Prior to Python 2.7, this function was not publicly documented. It is finally exposed publicly in Python 3.3 as the quote function in the shlex module.

Return a shell-escaped version of the string s. The returned value is a string that can safely be used as one token in a shell command line, for cases where you cannot use a list.

This idiom would be unsafe:
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print command  # executed by a shell: boom!
ls -l somefile; rm -rf ~

quote() lets you plug the security hole:

>>> command = 'ls -l {}'.format(quote(filename))
>>> print command
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print remote_command
ssh home 'ls -l 'somefile; rm -rf ~''

The quoting is compatible with UNIX shells and with shlex.split():

>>> remote_command = shlex.split(remote_command)
>>> remote_command
['ssh', 'home', 'ls -l 'somefile; rm -rf ~''
>>> command = shlex.split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
36.11.1. Template Objects

Template objects following methods:

Template.reset()

Restore a pipeline template to its initial state.

Template.clone()

Return a new, equivalent, pipeline template.

Template.debug(flag)

If flag is true, turn debugging on. Otherwise, turn debugging off. When debugging is on, commands to be executed are printed, and the shell is given `set -x` command to be more verbose.

Template.append(cmd, kind)

Append a new action at the end. The cmd variable must be a valid bourne shell command. The kind variable consists of two letters.

The first letter can be either of `-` (which means the command reads its standard input), `'f'` (which means the commands reads a given file on the command line) or `'.'` (which means the commands reads no input, and hence must be first.)

Similarly, the second letter can be either of `-` (which means the command writes to standard output), `'f'` (which means the command writes a file on the command line) or `'.'` (which means the command does not write anything, and hence must be last.)

Template.prepend(cmd, kind)

Add a new action at the beginning. See append() for explanations of the arguments.
Template. `open(file, mode)`

Return a file-like object, open to `file`, but read from or written to by the pipeline. Note that only one of `'r'`, `'w'` may be given.

Template. `copy(infile, outfile)`

Copy `infile` to `outfile` through the pipe.
36.12. **posixfile** — File-like objects with locking support

*Deprecated since version 1.5:* The locking operation that this module provides is done better and more portably by the `fcntl.lockf()` call.

This module implements some additional functionality over the built-in file objects. In particular, it implements file locking, control over the file flags, and an easy interface to duplicate the file object. The module defines a new file object, the posixfile object. It has all the standard file object methods and adds the methods described below. This module only works for certain flavors of Unix, since it uses `fcntl.fcntl()` for file locking.

To instantiate a posixfile object, use the `posixfile.open()` function. The resulting object looks and feels roughly the same as a standard file object.

The `posixfile` module defines the following constants:

```python
posixfile.SEEK_SET
  Offset is calculated from the start of the file.

posixfile.SEEK_CUR
  Offset is calculated from the current position in the file.

posixfile.SEEK_END
  Offset is calculated from the end of the file.
```

The `posixfile` module defines the following functions:

```python
posixfile.open(filename[, mode[, bufsize]])
```
Create a new posixfile object with the given filename and mode. The *filename*, *mode* and *bufsize* arguments are interpreted the same way as by the built-in *open()* function.

```python
posixfile.fileopen(fileobject)
```

Create a new posixfile object with the given standard file object. The resulting object has the same filename and mode as the original file object.

The posixfile object defines the following additional methods:

```python
posixfile.lock(fmt[, len[, start[, whence]]])
```

Lock the specified section of the file that the file object is referring to. The format is explained below in a table. The *len* argument specifies the length of the section that should be locked. The default is 0. *start* specifies the starting offset of the section, where the default is 0. The *whence* argument specifies where the offset is relative to. It accepts one of the constants `SEEK_SET`, `SEEK_CUR` or `SEEK_END`. The default is `SEEK_SET`. For more information about the arguments refer to the *fcntl(2)* manual page on your system.

```python
posixfile.flags([flags])
```

Set the specified flags for the file that the file object is referring to. The new flags are ORed with the old flags, unless specified otherwise. The format is explained below in a table. Without the *flags* argument a string indicating the current flags is returned (this is the same as the ? modifier). For more information about the flags refer to the *fcntl(2)* manual page on your system.

```python
posixfile.dup()
```

Duplicate the file object and the underlying file pointer and file descriptor. The resulting object behaves as if it were newly opened.
posixfile. **dup2(fd)**
Duplicate the file object and the underlying file pointer and file descriptor. The new object will have the given file descriptor. Otherwise the resulting object behaves as if it were newly opened.

posixfile. **file()**
Return the standard file object that the posixfile object is based on. This is sometimes necessary for functions that insist on a standard file object.

All methods raise **IOError** when the request fails.

Format characters for the **lock()** method have the following meaning:

<table>
<thead>
<tr>
<th>Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>unlock the specified region</td>
</tr>
<tr>
<td>r</td>
<td>request a read lock for the specified section</td>
</tr>
<tr>
<td>w</td>
<td>request a write lock for the specified section</td>
</tr>
</tbody>
</table>

In addition the following modifiers can be added to the format:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
<td>wait until the lock has been granted</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>return the first lock conflicting with the requested lock, or None if there is no conflict.</td>
<td>(1)</td>
</tr>
</tbody>
</table>

Note:

1. The lock returned is in the format *(mode, len, start, whence, pid)* where *mode* is a character representing the type of lock (*'r'* or *'w'*). This modifier prevents a request from being granted; it is for query purposes only.
Format characters for the `flags()` method have the following meanings:

<table>
<thead>
<tr>
<th>Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>append only flag</td>
</tr>
<tr>
<td>c</td>
<td>close on exec flag</td>
</tr>
<tr>
<td>n</td>
<td>no delay flag (also called non-blocking flag)</td>
</tr>
<tr>
<td>s</td>
<td>synchronization flag</td>
</tr>
</tbody>
</table>

In addition the following modifiers can be added to the format:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>turn the specified flags ‘off’, instead of the default ‘on’</td>
<td>(1)</td>
</tr>
<tr>
<td>=</td>
<td>replace the flags, instead of the default ‘OR’ operation</td>
<td>(1)</td>
</tr>
<tr>
<td>?</td>
<td>return a string in which the characters represent the flags</td>
<td>(2)</td>
</tr>
<tr>
<td></td>
<td>that are set.</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. The `!` and `=` modifiers are mutually exclusive.
2. This string represents the flags after they may have been altered by the same call.

Examples:

```python
import posixfile

def file = posixfile.open('testfile', 'w')
file.lock('w')
...
file.lock('u')
file.close()
```
36.13. resource — Resource usage information

This module provides basic mechanisms for measuring and controlling system resources utilized by a program.

Symbolic constants are used to specify particular system resources and to request usage information about either the current process or its children.

A single exception is defined for errors:

\textit{exception resource.error}

The functions described below may raise this error if the underlying system call failures unexpectedly.
36.13.1. Resource Limits

Resources usage can be limited using the `setrlimit()` function described below. Each resource is controlled by a pair of limits: a soft limit and a hard limit. The soft limit is the current limit, and may be lowered or raised by a process over time. The soft limit can never exceed the hard limit. The hard limit can be lowered to any value greater than the soft limit, but not raised. (Only processes with the effective UID of the super-user can raise a hard limit.)

The specific resources that can be limited are system dependent. They are described in the `getrlimit(2)` man page. The resources listed below are supported when the underlying operating system supports them; resources which cannot be checked or controlled by the operating system are not defined in this module for those platforms.

resource. **RLIM_INFINITY**

Constant used to represent the limit for an unlimited resource.

resource. **getrlimit(resource)**

Returns a tuple `(soft, hard)` with the current soft and hard limits of `resource`. Raises `ValueError` if an invalid resource is specified, or `error` if the underlying system call fails unexpectedly.

resource. **setrlimit(resource, limits)**

Sets new limits of consumption of `resource`. The `limits` argument must be a tuple `(soft, hard)` of two integers describing the new limits. A value of `RLIM_INFINITY` can be used to request a limit that is unlimited.

Raises `ValueError` if an invalid resource is specified, if the new soft limit exceeds the hard limit, or if a process tries to raise its
hard limit. Specifying a limit of `RLIM_INFINITY` when the hard or system limit for that resource is not unlimited will result in a `ValueError`. A process with the effective UID of super-user can request any valid limit value, including unlimited, but `ValueError` will still be raised if the requested limit exceeds the system imposed limit.

`setrlimit` may also raise `error` if the underlying system call fails.

These symbols define resources whose consumption can be controlled using the `setrlimit()` and `getrlimit()` functions described below. The values of these symbols are exactly the constants used by C programs.

The Unix man page for `getrlimit(2)` lists the available resources. Note that not all systems use the same symbol or same value to denote the same resource. This module does not attempt to mask platform differences — symbols not defined for a platform will not be available from this module on that platform.

**resource.** `RLIMIT_CORE`  
The maximum size (in bytes) of a core file that the current process can create. This may result in the creation of a partial core file if a larger core would be required to contain the entire process image.

**resource.** `RLIMIT_CPU`  
The maximum amount of processor time (in seconds) that a process can use. If this limit is exceeded, a `SIGXCPU` signal is sent to the process. (See the `signal` module documentation for information about how to catch this signal and do something useful, e.g. flush open files to disk.)

**resource.** `RLIMIT_FSIZE`  
The maximum size of a file which the process may create.
resource. **RLIMIT_DATA**  
The maximum size (in bytes) of the process's heap.

resource. **RLIMIT_STACK**  
The maximum size (in bytes) of the call stack for the current process. This only affects the stack of the main thread in a multi-threaded process.

resource. **RLIMIT_RSS**  
The maximum resident set size that should be made available to the process.

resource. **RLIMIT_NPROC**  
The maximum number of processes the current process may create.

resource. **RLIMIT_NOFILE**  
The maximum number of open file descriptors for the current process.

resource. **RLIMIT_OFILE**  
The BSD name for `RLIMIT_NOFILE`.

resource. **RLIMIT_MEMLOCK**  
The maximum address space which may be locked in memory.

resource. **RLIMIT_VMEM**  
The largest area of mapped memory which the process may occupy.

resource. **RLIMIT_AS**  
The maximum area (in bytes) of address space which may be taken by the process.
These functions are used to retrieve resource usage information:

```c
resource.getrusage(who)
```

This function returns an object that describes the resources consumed by either the current process or its children, as specified by the `who` parameter. The `who` parameter should be specified using one of the `RUSAGE_*` constants described below.

The fields of the return value each describe how a particular system resource has been used, e.g. amount of time spent running is user mode or number of times the process was swapped out of main memory. Some values are dependent on the clock tick internal, e.g. the amount of memory the process is using.

For backward compatibility, the return value is also accessible as a tuple of 16 elements.

The fields `ru_utime` and `ru_stime` of the return value are floating point values representing the amount of time spent executing in user mode and the amount of time spent executing in system mode, respectively. The remaining values are integers. Consult the `getrusage(2)` man page for detailed information about these values. A brief summary is presented here:

<table>
<thead>
<tr>
<th>Index</th>
<th>Field</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ru_utime</td>
<td>time in user mode (float)</td>
</tr>
<tr>
<td>1</td>
<td>ru_stime</td>
<td>time in system mode (float)</td>
</tr>
<tr>
<td>2</td>
<td>ru_maxrss</td>
<td>maximum resident set size</td>
</tr>
<tr>
<td>3</td>
<td>ru_ixrss</td>
<td>shared memory size</td>
</tr>
<tr>
<td>4</td>
<td>ru_idrss</td>
<td>unshared memory size</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>5</td>
<td><strong>ru_isrss</strong></td>
<td>unshared stack size</td>
</tr>
<tr>
<td>6</td>
<td><strong>ru_minflt</strong></td>
<td>page faults not requiring I/O</td>
</tr>
<tr>
<td>7</td>
<td><strong>ru_majflt</strong></td>
<td>page faults requiring I/O</td>
</tr>
<tr>
<td>8</td>
<td><strong>ru_nswap</strong></td>
<td>number of swap outs</td>
</tr>
<tr>
<td>9</td>
<td><strong>ru_inblock</strong></td>
<td>block input operations</td>
</tr>
<tr>
<td>10</td>
<td><strong>ru_oublock</strong></td>
<td>block output operations</td>
</tr>
<tr>
<td>11</td>
<td><strong>ru_msgsnd</strong></td>
<td>messages sent</td>
</tr>
<tr>
<td>12</td>
<td><strong>ru_msgrcv</strong></td>
<td>messages received</td>
</tr>
<tr>
<td>13</td>
<td><strong>ru_nsignals</strong></td>
<td>signals received</td>
</tr>
<tr>
<td>14</td>
<td><strong>ru_nvcsw</strong></td>
<td>voluntary context switches</td>
</tr>
<tr>
<td>15</td>
<td><strong>ru_nivcsw</strong></td>
<td>involuntary context switches</td>
</tr>
</tbody>
</table>

This function will raise a **ValueError** if an invalid **who** parameter is specified. It may also raise **error** exception in unusual circumstances.

*Changed in version 2.3:* Added access to values as attributes of the returned object.

**resource.**getpagesize()

Returns the number of bytes in a system page. (This need not be the same as the hardware page size.)

The following **RUSAGE_*** symbols are passed to the **getrusage()** function to specify which processes information should be provided for.

**resource.**RUSAGE_SELF

**RUSAGE_SELF** should be used to request information pertaining only to the process itself.

**resource.**RUSAGE_CHILDREN

Pass to **getrusage()** to request resource information for child processes of the calling process.
resource. **RUSAGE_BOTH**

Pass to `getrusage()` to request resources consumed by both the current process and child processes. May not be available on all systems.
The `nis` module gives a thin wrapper around the NIS library, useful for central administration of several hosts.

Because NIS exists only on Unix systems, this module is only available for Unix.

The `nis` module defines the following functions:

```python
nis.match(key, mapname[, domain=default_domain])
```

Return the match for `key` in map `mapname`, or raise an error (`nis.error`) if there is none. Both should be strings, `key` is 8-bit clean. Return value is an arbitrary array of bytes (may contain `NULL` and other joys).

Note that `mapname` is first checked if it is an alias to another name.

*Changed in version 2.5:* The `domain` argument allows overriding the NIS domain used for the lookup. If unspecified, lookup is in the default NIS domain.

```python
nis.cat(mapname[, domain=default_domain])
```

Return a dictionary mapping `key` to `value` such that `match(key, mapname)==value`. Note that both keys and values of the dictionary are arbitrary arrays of bytes.

Note that `mapname` is first checked if it is an alias to another name.

*Changed in version 2.5:* The `domain` argument allows overriding
the NIS domain used for the lookup. If unspecified, lookup is in
the default NIS domain.

```python
nis.maps([domain=default_domain])
```

Return a list of all valid maps.

*Changed in version 2.5:* The `domain` argument allows overriding
the NIS domain used for the lookup. If unspecified, lookup is in
the default NIS domain.

```python
nis.get_default_domain()
```

Return the system default NIS domain.

*New in version 2.5.*

The `nis` module defines the following exception:

```python
exception nis.error
```

An error raised when a NIS function returns an error code.
36.15. syslog — Unix syslog library routines

This module provides an interface to the Unix syslog library routines. Refer to the Unix manual pages for a detailed description of the syslog facility.

This module wraps the system syslog family of routines. A pure Python library that can speak to a syslog server is available in the logging.handlers module as SysLogHandler.

The module defines the following functions:

syslog. **syslog**(*message*)

syslog. **syslog**(*priority*, *message*)

Send the string message to the system logger. A trailing newline is added if necessary. Each message is tagged with a priority composed of a facility and a level. The optional priority argument, which defaults to LOG_INFO, determines the message priority. If the facility is not encoded in priority using logical-or (LOG_INFO | LOG_USER), the value given in the openlog() call is used.

If openlog() has not been called prior to the call to syslog(), openlog() will be called with no arguments.

syslog. **openlog**([*ident*[ [, *logoption*[ [, *facility*]]]])

Logging options of subsequent syslog() calls can be set by calling openlog(). syslog() will call openlog() with no arguments if the log is not currently open.

The optional ident keyword argument is a string which is prepended to every message, and defaults to sys.argv[0] with
leading path components stripped. The optional logoption keyword argument (default is 0) is a bit field – see below for possible values to combine. The optional facility keyword argument (default is LOG_USER) sets the default facility for messages which do not have a facility explicitly encoded.

**syslog.closelog()**
Reset the syslog module values and call the system library closelog().

This causes the module to behave as it does when initially imported. For example, openlog() will be called on the first syslog() call (if openlog() hasn't already been called), and ident and other openlog() parameters are reset to defaults.

**syslog.setlogmask(maskpri)**
Set the priority mask to maskpri and return the previous mask value. Calls to syslog() with a priority level not set in maskpri are ignored. The default is to log all priorities. The function LOG_MASK(pri) calculates the mask for the individual priority pri. The function LOG_UPTO(pri) calculates the mask for all priorities up to and including pri.

The module defines the following constants:

Priority levels (high to low):

- LOG_EMERG
- LOG_ALERT
- LOG_CRIT
- LOG_ERR
- LOG_WARNING
- LOG_NOTICE
- LOG_INFO
- LOG_DEBUG

Facilities:

- LOG_KERN
- LOG_USER
- LOG_MAIL
- LOG_DAEMON
- LOG_AUTH
- LOG_LPR
- LOG_NEWS
- LOG_UUCP
- LOG_CRON
- LOG_SYSLOG
- LOG_LOCAL0 to LOG_LOCAL7

Log options:
LOG_PID, LOG_CONS, LOG_NDELAY, LOG_NOWAIT and LOG_PERROR if defined in <syslog.h>. 
36.15.1. Examples

36.15.1.1. Simple example

A simple set of examples:

```python
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

An example of setting some log options, these would include the process ID in logged messages, and write the messages to the destination facility used for mail logging:

```python
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```
36.16. commands — Utilities for running commands

*Deprecated since version 2.6:* The commands module has been removed in Python 3. Use the subprocess module instead.

The commands module contains wrapper functions for os.popen() which take a system command as a string and return any output generated by the command and, optionally, the exit status.

The subprocess module provides more powerful facilities for spawning new processes and retrieving their results. Using the subprocess module is preferable to using the commands module.

**Note:** In Python 3.x, getstatus() and two undocumented functions (mk2arg() and mkarg()) have been removed. Also, getstatusoutput() and getoutput() have been moved to the subprocess module.

The commands module defines the following functions:

```python
commands.getstatusoutput(cmd)
```

Execute the string cmd in a shell with os.popen() and return a 2-tuple (status, output). cmd is actually run as `{ cmd ; } 2>&1`, so that the returned output will contain output or error messages. A trailing newline is stripped from the output. The exit status for the command can be interpreted according to the rules for the C function wait().

```python
commands.getoutput(cmd)
```
Like `getstatusoutput()`, except the exit status is ignored and the return value is a string containing the command's output.

```python
commands.getstatus(file)
```

Return the output of `ls -ld file` as a string. This function uses the `getoutput()` function, and properly escapes backslashes and dollar signs in the argument.

*Deprecated since version 2.6:* This function is nonobvious and useless. The name is also misleading in the presence of `getstatusoutput()`.

Example:

```python
>>> import commands
>>> commands.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> commands.getstatusoutput('cat /bin/junk')
(256, 'cat: /bin/junk: No such file or directory')
>>> commands.getstatusoutput('/bin/junk')
(256, 'sh: /bin/junk: not found')
>>> commands.getoutput('ls /bin/ls')
'/bin/ls'
>>> commands.getstatus('/bin/ls')
'-rwxr-xr-x  1 root  13352 Oct 14 1994 /bin/ls'
```

**See also:**

Module `subprocess`

Module for spawning and managing subprocesses.
37. Mac OS X specific services

This chapter describes modules that are only available on the Mac OS X platform.

See the chapters MacPython OSA Modules and Undocumented Mac OS modules for more modules, and the HOWTO Using Python on a Macintosh for a general introduction to Mac-specific Python programming.

Note: Most of the OS X APIs that these modules use are deprecated or removed in recent versions of OS X. Many are not available when Python is executing in 64-bit mode. These modules have been removed in Python 3. You should avoid using them in Python 2.

- 37.1. ic — Access to the Mac OS X Internet Config
  - 37.1.1. IC Objects
- 37.2. MacOS — Access to Mac OS interpreter features
- 37.3. macostools — Convenience routines for file manipulation
- 37.4. findertools — The finder's Apple Events interface
- 37.5. EasyDialogs — Basic Macintosh dialogs
  - 37.5.1. ProgressBar Objects
- 37.6. FrameWork — Interactive application framework
  - 37.6.1. Application Objects
  - 37.6.2. Window Objects
  - 37.6.3. ControlsWindow Object
  - 37.6.4. ScrolledWindow Object
  - 37.6.5. DialogWindow Objects
- 37.7. autoGIL — Global Interpreter Lock handling in event loops
- 37.8. Mac OS Toolbox Modules
  - 37.8.1. Carbon.AE — Apple Events
37.8.2. Carbon.AH — Apple Help
37.8.3. Carbon.App — Appearance Manager
37.8.4. Carbon.Appearance — Appearance Manager constants
37.8.5. Carbon.CF — Core Foundation
37.8.6. Carbon.CG — Core Graphics
37.8.7. Carbon.CarbonEvt — Carbon Event Manager
37.8.9. Carbon.Cm — Component Manager
37.8.10. Carbon.Components — Component Manager constants
37.8.11. Carbon.ControlAccessor — Control Manager accessors
37.8.12. Carbon.Controls — Control Manager constants
37.8.13. Carbon.CoreFoundation — CoreFoundation constants
37.8.15. Carbon.Ctrl — Control Manager
37.8.16. Carbon.Dialogs — Dialog Manager constants
37.8.17. Carbon.Dlg — Dialog Manager
37.8.18. Carbon.Drag — Drag and Drop Manager
37.8.19. Carbon.Dragconst — Drag and Drop Manager constants
37.8.20. Carbon.Events — Event Manager constants
37.8.22. Carbon.File — File Manager
37.8.23. Carbon.Files — File Manager constants
37.8.24. Carbon.Fm — Font Manager
37.8.25. Carbon.Folder — Folder Manager
37.8.27. Carbon.Fonts — Font Manager constants
37.8.28. **Carbon.Help** — Help Manager
37.8.29. **Carbon.IBCarbon** — Carbon InterfaceBuilder
37.8.30. **Carbon.IBCarbonRuntime** — Carbon InterfaceBuilder constants
37.8.31. **Carbon.Icn** — Carbon Icon Manager
37.8.32. **Carbon.Icons** — Carbon Icon Manager constants
37.8.33. **Carbon.Launch** — Carbon Launch Services
37.8.34. **Carbon.LaunchServices** — Carbon Launch Services constants
37.8.35. **Carbon.List** — List Manager
37.8.36. **Carbon.Lists** — List Manager constants
37.8.37. **Carbon.MacHelp** — Help Manager constants
37.8.38. **Carbon.MediaDescr** — Parsers and generators for Quicktime Media descriptors
37.8.39. **Carbon.Menu** — Menu Manager
37.8.40. **Carbon.Menus** — Menu Manager constants
37.8.41. **Carbon.Mlte** — MultiLingual Text Editor
37.8.42. **Carbon.OSA** — Carbon OSA Interface
37.8.43. **Carbon.OSAconst** — Carbon OSA Interface constants
37.8.44. **Carbon.QDOffscreen** — QuickDraw Offscreen constants
37.8.45. **Carbon.Qd** — QuickDraw
37.8.46. **Carbon.Qdoffs** — QuickDraw Offscreen
37.8.47. **Carbon.Qt** — QuickTime
37.8.48. **Carbon.QuickDraw** — QuickDraw constants
37.8.49. **Carbon.QuickTime** — QuickTime constants
37.8.50. **Carbon.Res** — Resource Manager and Handles
37.8.51. **Carbon.Resources** — Resource Manager and Handles constants
37.8.52. **Carbon.Scrap** — Scrap Manager
- 37.8.53. **Carbon.Snd** — Sound Manager
- 37.8.54. **Carbon.Sound** — Sound Manager constants
- 37.8.55. **Carbon.TE** — TextEdit
- 37.8.56. **Carbon.TextEdit** — TextEdit constants
- 37.8.57. **Carbon.Win** — Window Manager
- 37.8.58. **Carbon.Windows** — Window Manager constants
- 37.9. **ColorPicker** — Color selection dialog
37.1. **ic** — Access to the Mac OS X Internet Config

This module provides access to various internet-related preferences set through **System Preferences** or the **Finder**.

**Note:** This module has been removed in Python 3.x.

There is a low-level companion module **icglue** which provides the basic Internet Config access functionality. This low-level module is not documented, but the docstrings of the routines document the parameters and the routine names are the same as for the Pascal or C API to Internet Config, so the standard IC programmers’ documentation can be used if this module is needed.

The **ic** module defines the **error** exception and symbolic names for all error codes Internet Config can produce; see the source for details.

```python
exception ic.error
    Exception raised on errors in the ic module.
```

The **ic** module defines the following class and function:

```python
class ic.IC([signature[, ic]])
    Create an Internet Config object. The signature is a 4-character creator code of the current application (default 'Pyth') which may influence some of ICs settings. The optional ic argument is a low-level icglue.icinstance created beforehand, this may be useful if you want to get preferences from a different config file, etc.

ic.launchurl1(url[, hint])
```
ic.parseurl(data[, start[, end[, hint]]])
ic.mapfile(file)
ic.maptypecreator(type, creator[, filename])
ic.settypecreator(file)
    These functions are “shortcuts” to the methods of the same name, described below.
IC objects have a mapping interface, hence to obtain the mail address you simply get `ic['MailAddress']`. Assignment also works, and changes the option in the configuration file.

The module knows about various datatypes, and converts the internal IC representation to a “logical” Python data structure. Running the `ic` module standalone will run a test program that lists all keys and values in your IC database, this will have to serve as documentation.

If the module does not know how to represent the data it returns an instance of the `ICOpaqueData` type, with the raw data in its `data` attribute. Objects of this type are also acceptable values for assignment.

Besides the dictionary interface, IC objects have the following methods:

**IC.launchurl(url[, hint])**

Parse the given URL, launch the correct application and pass it the URL. The optional `hint` can be a scheme name such as `'mailto:'`, in which case incomplete URLs are completed with this scheme. If `hint` is not provided, incomplete URLs are invalid.

**IC.parseurl(data[, start[, end[, hint]]])**

Find a URL somewhere in `data` and return start position, end position and the URL. The optional `start` and `end` can be used to limit the search, so for instance if a user clicks in a long text field you can pass the whole text field and the click-position in `start` and this routine will return the whole URL in which the user clicked. As above, `hint` is an optional scheme used to complete
incomplete URLs.

**IC. mapfile(file)**

Return the mapping entry for the given file, which can be passed as either a filename or an \texttt{FSSpec()} result, and which need not exist.

The mapping entry is returned as a tuple \((version, \text{type}, \text{creator}, \text{postcreator}, \text{flags}, \text{extension}, \text{appname}, \text{postappname}, \text{mimetype}, \text{entryname})\), where \textit{version} is the entry version number, \textit{type} is the 4-character filetype, \textit{creator} is the 4-character creator type, \textit{postcreator} is the 4-character creator code of an optional application to post-process the file after downloading, \textit{flags} are various bits specifying whether to transfer in binary or ascii and such, \textit{extension} is the filename extension for this file type, \textit{appname} is the printable name of the application to which this file belongs, \textit{postappname} is the name of the postprocessing application, \textit{mimetype} is the MIME type of this file and \textit{entryname} is the name of this entry.

**IC. maptypecreator(type, creator[, filename])**

Return the mapping entry for files with given 4-character \textit{type} and \textit{creator} codes. The optional \textit{filename} may be specified to further help finding the correct entry (if the creator code is '????', for instance).

The mapping entry is returned in the same format as for \textit{mapfile}.

**IC. settypecreator(file)**

Given an existing \textit{file}, specified either as a filename or as an \texttt{FSSpec()} result, set its creator and type correctly based on its extension. The finder is told about the change, so the finder icon will be updated quickly.
This module provides access to MacOS specific functionality in the Python interpreter, such as how the interpreter eventloop functions and the like. Use with care.

**Note:** This module has been removed in Python 3.x.

Note the capitalization of the module name; this is a historical artifact.

**MacOS.runtimemodel**

Always 'macho', from Python 2.4 on. In earlier versions of Python the value could also be 'ppc' for the classic Mac OS 8 runtime model or 'carbon' for the Mac OS 9 runtime model.

**MacOS.linkmodel**

The way the interpreter has been linked. As extension modules may be incompatible between linking models, packages could use this information to give more decent error messages. The value is one of 'static' for a statically linked Python, 'framework' for Python in a Mac OS X framework, 'shared' for Python in a standard Unix shared library. Older Pythons could also have the value 'cfm' for Mac OS 9-compatible Python.

**exception MacOS.Error**

This exception is raised on MacOS generated errors, either from functions in this module or from other mac-specific modules like the toolbox interfaces. The arguments are the integer error code (the `OSErr` value) and a textual description of the error code. Symbolic names for all known error codes are defined in the
standard module `macerrors`.

**MacOS.** `GetErrorString(errno)`

Return the textual description of MacOS error code `errno`.

**MacOS.** `DebugStr(message[, object])`

On Mac OS X the string is simply printed to stderr (on older Mac OS systems more elaborate functionality was available), but it provides a convenient location to attach a breakpoint in a low-level debugger like `gdb`.

**Note:** Not available in 64-bit mode.

**MacOS.** `SysBeep()`

Ring the bell.

**Note:** Not available in 64-bit mode.

**MacOS.** `GetTicks()`

Get the number of clock ticks (1/60th of a second) since system boot.

**MacOS.** `GetCreatorAndType(file)`

Return the file creator and file type as two four-character strings. The `file` parameter can be a pathname or an `FSSpec` or `FSRef` object.

**Note:** It is not possible to use an `FSSpec` in 64-bit mode.

**MacOS.** `SetCreatorAndType(file, creator, type)`

Set the file creator and file type. The `file` parameter can be a pathname or an `FSSpec` or `FSRef` object. `creator` and `type` must be four character strings.
Note: It is not possible to use an FSSpec in 64-bit mode.

MacOS. `openrf(name[, mode])`

Open the resource fork of a file. Arguments are the same as for the built-in function `open()`. The object returned has file-like semantics, but it is not a Python file object, so there may be subtle differences.

MacOS. `WMAvailable()`

Checks whether the current process has access to the window manager. The method will return `False` if the window manager is not available, for instance when running on Mac OS X Server or when logged in via ssh, or when the current interpreter is not running from a fullblown application bundle. A script runs from an application bundle either when it has been started with `pythonw` instead of `python` or when running as an applet.

MacOS. `splash([resourceid])`

Opens a splash screen by resource id. Use `resourceid θ` to close the splash screen.

Note: Not available in 64-bit mode.
This module contains some convenience routines for file-manipulation on the Macintosh. All file parameters can be specified as pathnames, `FSRef` or `FSSpec` objects. This module expects a filesystem which supports forked files, so it should not be used on UFS partitions.

**Note:** This module has been removed in Python 3.

The `macostools` module defines the following functions:

### `macostools.copy(src, dst[, createpath[, copytimes]])`
Copy file `src` to `dst`. If `createpath` is non-zero the folders leading to `dst` are created if necessary. The method copies data and resource fork and some finder information (creator, type, flags) and optionally the creation, modification and backup times (default is to copy them). Custom icons, comments and icon position are not copied.

**Note:** This function does not work in 64-bit code because it uses APIs that are not available in 64-bit mode.

### `macostools.copytree(src, dst)`
Recursively copy a file tree from `src` to `dst`, creating folders as needed. `src` and `dst` should be specified as pathnames.

**Note:** This function does not work in 64-bit code because it uses APIs that are not available in 64-bit mode.

### `macostools.mkalias(src, dst)`
Create a finder alias \textit{dst} pointing to \textit{src}.

\textbf{Note:} This function does not work in 64-bit code because it uses APIs that are not available in 64-bit mode.

\texttt{macostools.touched(\textit{dst})}

Tell the finder that some bits of finder-information such as creator or type for file \textit{dst} has changed. The file can be specified by pathname or fsspec. This call should tell the finder to redraw the files icon.

\textcolor{red}{\textit{Deprecated since version 2.6:} The function is a no-op on OS X.}

\texttt{macostools.BUFSIZ}

The buffer size for \texttt{copy}, default 1 megabyte.

Note that the process of creating finder aliases is not specified in the Apple documentation. Hence, aliases created with \texttt{mkalias()} could conceivably have incompatible behaviour in some cases.
This module contains routines that give Python programs access to some functionality provided by the finder. They are implemented as wrappers around the AppleEvent interface to the finder.

All file and folder parameters can be specified either as full pathnames, or as FSRef or FSSpec objects.

The findertools module defines the following functions:

findertools.launch(file)
Tell the finder to launch file. What launching means depends on the file: applications are started, folders are opened and documents are opened in the correct application.

findertools.Print(file)
Tell the finder to print a file. The behaviour is identical to selecting the file and using the print command in the finder’s file menu.

findertools.copy(file, destdir)
Tell the finder to copy a file or folder file to folder destdir. The function returns an Alias object pointing to the new file.

findertools.move(file, destdir)
Tell the finder to move a file or folder file to folder destdir. The function returns an Alias object pointing to the new file.

findertools.sleep()
Tell the finder to put the Macintosh to sleep, if your machine supports it.

findertools.restart()
Tell the finder to perform an orderly restart of the machine.

```
findertools.shutdown()
```

Tell the finder to perform an orderly shutdown of the machine.
37.5. EasyDialogs — Basic Macintosh dialogs

The EasyDialogs module contains some simple dialogs for the Macintosh. The dialogs get launched in a separate application which appears in the dock and must be clicked on for the dialogs be displayed. All routines take an optional resource ID parameter id with which one can override the DLOG resource used for the dialog, provided that the dialog items correspond (both type and item number) to those in the default DLOG resource. See source code for details.

**Note:** This module has been removed in Python 3.x.

The EasyDialogs module defines the following functions:

**EasyDialogs.**`

- `Message(str[, id[, ok]])`
  Displays a modal dialog with the message text `str`, which should be at most 255 characters long. The button text defaults to “OK”, but is set to the string argument `ok` if the latter is supplied. Control is returned when the user clicks the “OK” button.

- `AskString(prompt[, default[, id[, ok[, cancel]]]])`
  Asks the user to input a string value via a modal dialog. `prompt` is the prompt message, and the optional `default` supplies the initial value for the string (otherwise an empty string is used). The text of the “OK” and “Cancel” buttons can be changed with the `ok` and `cancel` arguments. All strings can be at most 255 bytes long. `AskString()` returns the string entered or `None` in case the user cancelled.
EasyDialogs. **AskPassword**(prompt[, default[, id[, ok[, cancel]]]]))

Asks the user to input a string value via a modal dialog. Like **AskString()**, but with the text shown as bullets. The arguments have the same meaning as for **AskString()**.

EasyDialogs. **AskYesNoCancel**(question[, default[, yes[, no[, cancel[, id]]]]])

Presents a dialog with prompt question and three buttons labelled “Yes”, “No”, and “Cancel”. Returns 1 for “Yes”, 0 for “No” and -1 for “Cancel”. The value of default (or 0 if default is not supplied) is returned when the RETURN key is pressed. The text of the buttons can be changed with the yes, no, and cancel arguments; to prevent a button from appearing, supply ”” for the corresponding argument.

EasyDialogs. **ProgressBar**([title[, maxval[, label[, id]]]]))

Displays a modeless progress-bar dialog. This is the constructor for the **ProgressBar** class described below. title is the text string displayed (default “Working...”), maxval is the value at which progress is complete (default 0, indicating that an indeterminate amount of work remains to be done), and label is the text that is displayed above the progress bar itself.

EasyDialogs. **GetArgv**(optionlist[ commandlist[, addoldfile[, addnewfile[, addfolder[, id]]]]])

Displays a dialog which aids the user in constructing a command-line argument list. Returns the list in sys.argv format, suitable for passing as an argument to getopt.getopt(). addoldfile, addnewfile, and addfolder are boolean arguments. When nonzero, they enable the user to insert into the command line paths to an existing file, a (possibly) not-yet-existent file, and a folder, respectively. (Note: Option arguments must appear in the
command line before file and folder arguments in order to be recognized by `getopt.getopt()`.) Arguments containing spaces can be specified by enclosing them within single or double quotes. A `SystemExit` exception is raised if the user presses the “Cancel” button.

`optionlist` is a list that determines a popup menu from which the allowed options are selected. Its items can take one of two forms: `optstr` or `(optstr, descr)`. When present, `descr` is a short descriptive string that is displayed in the dialog while this option is selected in the popup menu. The correspondence between `optstrs` and command-line arguments is:

<table>
<thead>
<tr>
<th><code>optstr</code> format</th>
<th>Command-line format</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-x (short option)</td>
</tr>
<tr>
<td>x: or x=</td>
<td>-x (short option with value)</td>
</tr>
<tr>
<td>xyz</td>
<td>--xyz (long option)</td>
</tr>
<tr>
<td>xyz: or xyz=</td>
<td>--xyz (long option with value)</td>
</tr>
</tbody>
</table>

`commandlist` is a list of items of the form `cmdstr` or `(cmdstr, descr)`, where `descr` is as above. The `cmdstrs` will appear in a popup menu. When chosen, the text of `cmdstr` will be appended to the command line as is, except that a trailing `':='` or `'= '` (if present) will be trimmed off.

*New in version 2.0.*


Post a dialog asking the user for a file to open, and return the file selected or `None` if the user cancelled. `message` is a text
message to display, typeList is a list of 4-char filetypes allowable, defaultLocation is the pathname, FSSpec or FSRef of the folder to show initially, location is the \((x, y)\) position on the screen where the dialog is shown, actionButtonLabel is a string to show instead of “Open” in the OK button, cancelButtonLabel is a string to show instead of “Cancel” in the cancel button, wanted is the type of value wanted as a return: str, unicode, FSSpec, FSRef and subtypes thereof are acceptable.

For a description of the other arguments please see the Apple Navigation Services documentation and the EasyDialogs source code.


Post a dialog asking the user for a file to save to, and return the file selected or **None** if the user cancelled. savedFileName is the default for the file name to save to (the return value). See AskFileForOpen() for a description of the other arguments.


Post a dialog asking the user to select a folder, and return the folder selected or **None** if the user cancelled. See AskFileForOpen() for a description of the arguments.

**See also:**

**Navigation Services Reference**

Programmer's reference documentation for the Navigation
Services, a part of the Carbon framework.
37.5.1. ProgressBar Objects

ProgressBar objects provide support for modeless progress-bar dialogs. Both determinate (thermometer style) and indeterminate (barber-pole style) progress bars are supported. The bar will be determinate if its maximum value is greater than zero; otherwise it will be indeterminate.

*Changed in version 2.2:* Support for indeterminate-style progress bars was added.

The dialog is displayed immediately after creation. If the dialog’s “Cancel” button is pressed, or if `Cmd-.` or `ESC` is typed, the dialog window is hidden and `KeyboardInterrupt` is raised (but note that this response does not occur until the progress bar is next updated, typically via a call to `inc()` or `set()`). Otherwise, the bar remains visible until the ProgressBar object is discarded.

ProgressBar objects possess the following attributes and methods:

**ProgressBar.** `curval`

The current value (of type integer or long integer) of the progress bar. The normal access methods coerce `curval` between 0 and `maxval`. This attribute should not be altered directly.

**ProgressBar.** `maxval`

The maximum value (of type integer or long integer) of the progress bar; the progress bar (thermometer style) is full when `curval` equals `maxval`. If `maxval` is 0, the bar will be indeterminate (barber-pole). This attribute should not be altered directly.

**ProgressBar.** `title([newstr])`

Sets the text in the title bar of the progress dialog to `newstr`. 
ProgressBar. **label**([*newstr*])  
Sets the text in the progress box of the progress dialog to *newstr*.

ProgressBar. **set**(*value*, *max*)  
Sets the progress bar’s *curval* to *value*, and also *maxval* to *max* if the latter is provided. *value* is first coerced between 0 and *maxval*. The thermometer bar is updated to reflect the changes, including a change from indeterminate to determinate or vice versa.

ProgressBar. **inc**(*n*)  
Increments the progress bar’s *curval* by *n*, or by 1 if *n* is not provided. (Note that *n* may be negative, in which case the effect is a decrement.) The progress bar is updated to reflect the change. If the bar is indeterminate, this causes one “spin” of the barber pole. The resulting *curval* is coerced between 0 and *maxval* if incrementing causes it to fall outside this range.
37.6. FrameWork — Interactive application framework

The FrameWork module contains classes that together provide a framework for an interactive Macintosh application. The programmer builds an application by creating subclasses that override various methods of the bases classes, thereby implementing the functionality wanted. Overriding functionality can often be done on various different levels, i.e. to handle clicks in a single dialog window in a non-standard way it is not necessary to override the complete event handling.

**Note:** This module has been removed in Python 3.x.

Work on the FrameWork has pretty much stopped, now that PyObjc is available for full Cocoa access from Python, and the documentation describes only the most important functionality, and not in the most logical manner at that. Examine the source or the examples for more details. The following are some comments posted on the MacPython newsgroup about the strengths and limitations of FrameWork:

The strong point of FrameWork is that it allows you to break into the control-flow at many different places. w, for instance, uses a different way to enable/disable menus and that plugs right in leaving the rest intact. The weak points of FrameWork are that it has no abstract command interface (but that shouldn’t be difficult), that its dialog support is minimal and that its control/toolbar support is non-existent.

The FrameWork module defines the following functions:

FrameWork.Application()
An object representing the complete application. See below for a description of the methods. The default \texttt{__init__()} routine creates an empty window dictionary and a menu bar with an apple menu.

\texttt{FrameWork.MenuBar()}  
An object representing the menubar. This object is usually not created by the user.

\texttt{FrameWork.Menu(bar, title[, after])}  
An object representing a menu. Upon creation you pass the \texttt{MenuBar} the menu appears in, the \texttt{title} string and a position (1-based) \texttt{after} where the menu should appear (default: at the end).

\texttt{FrameWork.MenuItem(menu, title[, shortcut, callback])}  
Create a menu item object. The arguments are the menu to create, the item title string and optionally the keyboard shortcut and a callback routine. The callback is called with the arguments menu-id, item number within menu (1-based), current front window and the event record.

Instead of a callable object the callback can also be a string. In this case menu selection causes the lookup of a method in the topmost window and the application. The method name is the callback string with \texttt{'domenu_'} prepended.

Calling the \texttt{MenuBar fixmenudimstate()} method sets the correct dimming for all menu items based on the current front window.

\texttt{FrameWork.Separator(menu)}  
Add a separator to the end of a menu.

\texttt{FrameWork.SubMenu(menu, label)}  
Create a submenu named \texttt{label} under menu \texttt{menu}. The menu
object is returned.

**Framework.** `Window(parent)`

Creates a (modeless) window. *Parent* is the application object to which the window belongs. The window is not displayed until later.

**Framework.** `DialogWindow(parent)`

Creates a modeless dialog window.

**Framework.** `windowbounds(width, height)`

Return a *(left, top, right, bottom)* tuple suitable for creation of a window of given width and height. The window will be staggered with respect to previous windows, and an attempt is made to keep the whole window on-screen. However, the window will however always be the exact size given, so parts may be offscreen.

**Framework.** `setwatchescursor()`

Set the mouse cursor to a watch.

**Framework.** `setarrowcursor()`

Set the mouse cursor to an arrow.
37.6.1. Application Objects

Application objects have the following methods, among others:

Application.makeusermenus()
Override this method if you need menus in your application. Append the menus to the attribute menubar.

Application.getabouttext()
Override this method to return a text string describing your application. Alternatively, override the do_about() method for more elaborate “about” messages.

Application.mainloop([mask[, wait]])
This routine is the main event loop, call it to set your application rolling. Mask is the mask of events you want to handle, wait is the number of ticks you want to leave to other concurrent application (default 0, which is probably not a good idea). While raising self to exit the mainloop is still supported it is not recommended: call self._quit() instead.

The event loop is split into many small parts, each of which can be overridden. The default methods take care of dispatching events to windows and dialogs, handling drags and resizes, Apple Events, events for non-FrameWork windows, etc.

In general, all event handlers should return 1 if the event is fully handled and 0 otherwise (because the front window was not a FrameWork window, for instance). This is needed so that update events and such can be passed on to other windows like the Sioux console window. Calling MacOS.HandleEvent() is not allowed within our dispatch or its callees, since this may result in an infinite loop if the code is called through the Python inner-loop
event handler.

Application. **asyncevents(onoff)**

Call this method with a nonzero parameter to enable asynchronous event handling. This will tell the inner interpreter loop to call the application event handler *async_dispatch* whenever events are available. This will cause FrameWork window updates and the user interface to remain working during long computations, but will slow the interpreter down and may cause surprising results in non-reentrant code (such as FrameWork itself). By default *async_dispatch* will immediately call *our_dispatch* but you may override this to handle only certain events asynchronously. Events you do not handle will be passed to Sioux and such.

The old on/off value is returned.

Application. **_quit()**

Terminate the running *mainloop()* call at the next convenient moment.

Application. **do_char(c, event)**

The user typed character *c*. The complete details of the event can be found in the *event* structure. This method can also be provided in a *Window* object, which overrides the application-wide handler if the window is frontmost.

Application. **do_dialogevent(event)**

Called early in the event loop to handle modeless dialog events. The default method simply dispatches the event to the relevant dialog (not through the *DialogWindow* object involved). Override if you need special handling of dialog events (keyboard shortcuts, etc).

Application. **idle(event)**
Called by the main event loop when no events are available. The null-event is passed (so you can look at mouse position, etc).
37.6.2. Window Objects

Window objects have the following methods, among others:

Window. **open()**
Override this method to open a window. Store the Mac OS window-id in `self.wid` and call the `do_postopen()` method to register the window with the parent application.

Window. **close()**
Override this method to do any special processing on window close. Call the `do_postclose()` method to cleanup the parent state.

Window. **do_postresize(width, height, macoswindowid)**
Called after the window is resized. Override if more needs to be done than calling `InvalRect`.

Window. **do_contentclick(local, modifiers, event)**
The user clicked in the content part of a window. The arguments are the coordinates (window-relative), the key modifiers and the raw event.

Window. **do_update(macoswindowid, event)**
An update event for the window was received. Redraw the window.

Window. **do_activate(activate, event)**
The window was activated (`activate == 1`) or deactivated (`activate == 0`). Handle things like focus highlighting, etc.
37.6.3. ControlsWindow Object

ControlsWindow objects have the following methods besides those of Window objects:

ControlsWindow.\texttt{do\_controlhit}(window, control, pcode, event)

Part \textit{pcode} of control \textit{control} was hit by the user. Tracking and such has already been taken care of.
37.6.4. ScrolledWindow Object

ScrolledWindow objects are ControlsWindow objects with the following extra methods:

**ScrolledWindow.** `scrollbars([wantx[, wanty]])`

Create (or destroy) horizontal and vertical scrollbars. The arguments specify which you want (default: both). The scrollbars always have minimum 0 and maximum 32767.

**ScrolledWindow.** `getscrollbarvalues()`

You must supply this method. It should return a tuple `(x, y)` giving the current position of the scrollbars (between 0 and 32767). You can return `None` for either to indicate the whole document is visible in that direction.

**ScrolledWindow.** `updatescrollbars()`

Call this method when the document has changed. It will call `getscrollbarvalues()` and update the scrollbars.

**ScrolledWindow.** `scrollbar_callback(which, what, value)`

Supplied by you and called after user interaction. `which` will be 'x' or 'y', `what` will be '-', '--', 'set', '++' or '+'. For 'set', `value` will contain the new scrollbar position.

**ScrolledWindow.** `scalebarvalues(absmin, absmax, curmin, curmax)`

Auxiliary method to help you calculate values to return from `getscrollbarvalues()`. You pass document minimum and maximum value and topmost (leftmost) and bottommost (rightmost) visible values and it returns the correct number or `None`.

**ScrolledWindow.** `do_activate(onoff, event)`
Takes care of dimming/highlighting scrollbars when a window becomes frontmost. If you override this method, call this one at the end of your method.

ScrolledWindow. **do_postresize**(width, height, window)

Moves scrollbars to the correct position. Call this method initially if you override it.

ScrolledWindow. **do_controlhit**(window, control, pcode, event)

Handles scrollbar interaction. If you override it call this method first, a nonzero return value indicates the hit was in the scrollbars and has been handled.
37.6.5. DialogWindow Objects

DialogWindow objects have the following methods besides those of Window objects:

**DialogWindow.open(resid)**

Create the dialog window, from the DLOG resource with id `resid`. The dialog object is stored in `self.wid`.

**DialogWindow.do_itemhit(item, event)**

Item number `item` was hit. You are responsible for redrawing toggle buttons, etc.
The `autoGIL` module provides a function `installAutoGIL()` that automatically locks and unlocks Python’s Global Interpreter Lock when running an event loop.

**Note:** This module has been removed in Python 3.x.

```python
exception autoGIL.AutoGILError
    Raised if the observer callback cannot be installed, for example because the current thread does not have a run loop.
```

```python
autoGIL.installAutoGIL()
    Install an observer callback in the event loop (CFRunLoop) for the current thread, that will lock and unlock the Global Interpreter Lock (GIL) at appropriate times, allowing other Python threads to run while the event loop is idle.
```

*Availability:* OSX 10.1 or later.
37.8. Mac OS Toolbox Modules

These are a set of modules that provide interfaces to various legacy Mac OS toolboxes. If applicable the module will define a number of Python objects for the various structures declared by the toolbox, and operations will be implemented as methods of the object. Other operations will be implemented as functions in the module. Not all operations possible in C will also be possible in Python (callbacks are often a problem), and parameters will occasionally be different in Python (input and output buffers, especially). All methods and functions have a \_doc\_ string describing their arguments and return values, and for additional description you are referred to Inside Macintosh or similar works.

These modules all live in a package called \texttt{carbon}. Despite that name they are not all part of the Carbon framework: CF is really in the CoreFoundation framework and Qt is in the QuickTime framework. The normal use pattern is

\begin{verbatim}
from Carbon import AE
\end{verbatim}

\textbf{Note:} Most of the OS X APIs that these modules use are deprecated or removed in recent versions of OS X. Many are not available when Python is executing in 64-bit mode. The Carbon modules have been removed in Python 3. You should avoid using them in Python 2.
37.8.1. Carbon.AE — Apple Events
37.8.3. Carbon.App — Appearance Manager
37.8.4. Carbon.Appearance — Appearance Manager constants
37.8.5. Carbon.CF — Core Foundation

The CFBase, CFArray, CFData, CFDictionary, CFString and CFURL objects are supported, some only partially.
37.8.7. Carbon.CarbonEvt — Carbon Event Manager
37.8.9. Carbon.Cm — Component Manager
37.8.11. Carbon.Control1Accessor — Control Manager accessors
37.8.12. Carbon.Controls — Control Manager constants
37.8.13. Carbon.CoreFoundation — CoreFoundation constants
Carbon.Dialogs — Dialog Manager constants
37.8.17. Carbon.Dlg — Dialog Manager
37.8.18. CarbonDrag — Drag and Drop Manager
37.8.19. **Carbon.Dragconst** — Drag and Drop Manager constants
37.8.20. *Carbon.Events* — Event Manager constants
37.8.22. Carbon.File — File Manager
37.8.23. Carbon.Files — File Manager constants
37.8.25. Carbon.Folder — Folder Manager
37.8.26. **Carbon.Folders** — Folder Manager constants
37.8.27. *Carbon.Fonts* — Font Manager constants
37.8.28. Carbon.Help — Help Manager
37.8.29. `Carbon.IBCarbon` — Carbon InterfaceBuilder
37.8.30. `Carbon.IBCarbonRuntime` — Carbon InterfaceBuilder constants
37.8.31. Carbon.Icon — Carbon Icon Manager
37.8.32. Carbon::Icons — Carbon Icon Manager constants
37.8.33. Carbon Launch — Carbon Launch Services
37.8.34. Carbon.LaunchServices — Carbon Launch Services constants
37.8.35. Carbon.List — List Manager
37.8.36. Carbon.Lists — List Manager constants
37.8.37. Carbon.MacHelp — Help Manager constants
37.8.38. Carbon.MediaDescr — Parsers and generators for Quicktime Media descriptors
37.8.40. *Carbon.Menus* — Menu Manager constants
37.8.41. Carbon.Mlte — MultiLingual Text Editor
37.8.42. Carbon OSA — Carbon OSA Interface
37.8.43. **Carbon.OSAconst** — Carbon OSA Interface constants
37.8.44. Carbon.QDOffscreen — QuickDraw Offscreen constants
37.8.46. Carbon.Qdoffs — QuickDraw Offscreen
| 37.8.47. Carbon.Qt — QuickTime |
37.8.48. Carbon.QuickDraw — QuickDraw constants
37.8.49. **Carbon.QuickTime** — **QuickTime constants**
37.8.51. Carbon.Resources — Resource Manager and Handles constants
This module is only fully available on Mac OS 9 and earlier under classic PPC MacPython. Very limited functionality is available under Carbon MacPython.

The Scrap Manager supports the simplest form of cut & paste operations on the Macintosh. It can be use for both inter- and intra-application clipboard operations.

The scrap module provides low-level access to the functions of the Scrap Manager. It contains the following functions:

```
Carbon.Scrap.InfoScrap()
```

Return current information about the scrap. The information is encoded as a tuple containing the fields (size, handle, count, state, path).

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>Size of the scrap in bytes.</td>
</tr>
<tr>
<td>handle</td>
<td>Resource object representing the scrap.</td>
</tr>
<tr>
<td>count</td>
<td>Serial number of the scrap contents.</td>
</tr>
<tr>
<td>state</td>
<td>Integer; positive if in memory, 0 if on disk, negative if uninitialized.</td>
</tr>
<tr>
<td>path</td>
<td>Filename of the scrap when stored on disk.</td>
</tr>
</tbody>
</table>

**See also:**

Scrap Manager

Apple’s documentation for the Scrap Manager gives a lot of useful information about using the Scrap Manager in applications.
37.8.54. `Carbon.Sound` — Sound Manager constants
37.8.55. Carbon. TE — TextEdit
37.8.56. Carbon::TextEdit — TextEdit

constants
37.9. ColorPicker — Color selection dialog

The ColorPicker module provides access to the standard color picker dialog.

**Note:** This module has been removed in Python 3.x.

ColorPicker. **getColor**(prompt, rgb)

Show a standard color selection dialog and allow the user to select a color. The user is given instruction by the prompt string, and the default color is set to rgb. rgb must be a tuple giving the red, green, and blue components of the color. GetColor() returns a tuple giving the user’s selected color and a flag indicating whether they accepted the selection of cancelled.
This chapter describes the current implementation of the Open Scripting Architecture (OSA, also commonly referred to as AppleScript) for Python, allowing you to control scriptable applications from your Python program, and with a fairly pythonic interface. Development on this set of modules has stopped.

For a description of the various components of AppleScript and OSA, and to get an understanding of the architecture and terminology, you should read Apple’s documentation. The “Applescript Language Guide” explains the conceptual model and the terminology, and documents the standard suite. The “Open Scripting Architecture” document explains how to use OSA from an application programmers point of view. In the Apple Help Viewer these books are located in the Developer Documentation, Core Technologies section.

As an example of scripting an application, the following piece of AppleScript will get the name of the frontmost Finder window and print it:

```
tell application "Finder"
    get name of window 1
end tell
```

In Python, the following code fragment will do the same:

```
import Finder

f = Finder.Finder()
print f.get(f.window(1).name)
```

As distributed the Python library includes packages that implement the standard suites, plus packages that interface to a small number of common applications.
To send AppleEvents to an application you must first create the Python package interfacing to the terminology of the application (what Script Editor calls the “Dictionary”). This can be done from within the PythonIDE or by running the gensuitemodule.py module as a standalone program from the command line.

The generated output is a package with a number of modules, one for every suite used in the program plus an __init__ module to glue it all together. The Python inheritance graph follows the AppleScript inheritance graph, so if a program’s dictionary specifies that it includes support for the Standard Suite, but extends one or two verbs with extra arguments then the output suite will contain a module Standard_Suite that imports and re-exports everything from StdSuites.Standard_Suite but overrides the methods that have extra functionality. The output of gensuitemodule is pretty readable, and contains the documentation that was in the original AppleScript dictionary in Python docstrings, so reading it is a good source of documentation.

The output package implements a main class with the same name as the package which contains all the AppleScript verbs as methods, with the direct object as the first argument and all optional parameters as keyword arguments. AppleScript classes are also implemented as Python classes, as are comparisons and all the other thingies.

The main Python class implementing the verbs also allows access to the properties and elements declared in the AppleScript class “application”. In the current release that is as far as the object orientation goes, so in the example above we need to use f.get(f.window(1).name) instead of the more Pythonic f.window(1).name.get().

If an AppleScript identifier is not a Python identifier the name is
mangled according to a small number of rules:

- spaces are replaced with underscores
- other non-alphanumeric characters are replaced with \_\text{xx}\_ where \text{xx} is the hexadecimal character value
- any Python reserved word gets an underscore appended

Python also has support for creating scriptable applications in Python, but The following modules are relevant to MacPython AppleScript support:

- 38.1. \texttt{gensuitemodule} — Generate OSA stub packages
- 38.2. \texttt{aetools} — OSA client support
- 38.3. \texttt{aepack} — Conversion between Python variables and AppleEvent data containers
- 38.4. \texttt{aetypes} — AppleEvent objects
- 38.5. \texttt{MiniAEFrame} — Open Scripting Architecture server support
  - 38.5.1. \texttt{AEServer Objects}

In addition, support modules have been pre-generated for Finder, Terminal, Explorer, Netscape, CodeWarrior, SystemEvents and StdSuites.
38.1. gensuitemodule — Generate OSA stub packages

The `gensuitemodule` module creates a Python package implementing stub code for the AppleScript suites that are implemented by a specific application, according to its AppleScript dictionary.

It is usually invoked by the user through the PythonIDE, but it can also be run as a script from the command line (pass `--help` for help on the options) or imported from Python code. For an example of its use see `Mac/scripts/genallsuites.py` in a source distribution, which generates the stub packages that are included in the standard library.

It defines the following public functions:

- `gensuitemodule.is_scriptable(application)`
  Returns true if `application`, which should be passed as a pathname, appears to be scriptable. Take the return value with a grain of salt: Internet Explorer appears not to be scriptable but definitely is.

- `gensuitemodule.processfile(application[, output, basepkgname, edit_modnames, creatorsignature, dump, verbose])`
  Create a stub package for `application`, which should be passed as a full pathname. For a `.app` bundle this is the pathname to the bundle, not to the executable inside the bundle; for an unbundled CFM application you pass the filename of the application binary.

  This function asks the application for its OSA terminology resources, decodes these resources and uses the resultant data to create the Python code for the package implementing the
client stubs.

output is the pathname where the resulting package is stored, if not specified a standard “save file as” dialog is presented to the user. basepkgname is the base package on which this package will build, and defaults to stdsuites. Only when generating stdsuites itself do you need to specify this. edit_modnames is a dictionary that can be used to change modulenames that are too ugly after name mangling. creator_signature can be used to override the 4-char creator code, which is normally obtained from the PkgInfo file in the package or from the CFM file creator signature. When dump is given it should refer to a file object, and processfile will stop after decoding the resources and dump the Python representation of the terminology resources to this file. verbose should also be a file object, and specifying it will cause processfile to tell you what it is doing.

gensuitemodule.processfile_fromresource(application[, output, basepkgname, edit_modnames, creatorsignature, dump, verbose])

This function does the same as processfile, except that it uses a different method to get the terminology resources. It opens application as a resource file and reads all "aete" and "aeut" resources from this file.
38.2. aetools — OSA client support

The aetools module contains the basic functionality on which Python AppleScript client support is built. It also imports and re-exports the core functionality of the aetypes and aepack modules. The stub packages generated by gensuitemodule import the relevant portions of aetools, so usually you do not need to import it yourself. The exception to this is when you cannot use a generated suite package and need lower-level access to scripting.

The aetools module itself uses the AppleEvent support provided by the Carbon.AE module. This has one drawback: you need access to the window manager, see section Running scripts with a GUI for details. This restriction may be lifted in future releases.

**Note:** This module has been removed in Python 3.x.

The aetools module defines the following functions:

- **aetools.packevent** *(ae, parameters, attributes)*
  Stores parameters and attributes in a pre-created Carbon.AE.AEDesc object. parameters and attributes are dictionaries mapping 4-character OSA parameter keys to Python objects. The objects are packed using aepack.pack().

- **aetools.unpackevent** *(ae[, formodulename])*
  Recursively unpacks a Carbon.AE.AEDesc event to Python objects. The function returns the parameter dictionary and the attribute dictionary. The formodulename argument is used by generated stub packages to control where AppleScript classes are looked up.
aetools. **keystub** *(arguments, keydict)*  
Converts a Python keyword argument dictionary `arguments` to the format required by `packevent` by replacing the keys, which are Python identifiers, by the four-character OSA keys according to the mapping specified in `keydict`. Used by the generated suite packages.

aetools. **enumsubst** *(arguments, key, edict)*  
If the `arguments` dictionary contains an entry for `key` convert the value for that entry according to dictionary `edict`. This converts human-readable Python enumeration names to the OSA 4-character codes. Used by the generated suite packages.

The *aetools* module defines the following class:

```python
class aetools.TalkTo([signature=None, start=0, timeout=0])
```

Base class for the proxy used to talk to an application. `signature` overrides the class attribute `_signature` (which is usually set by subclasses) and is the 4-char creator code defining the application to talk to. `start` can be set to true to enable running the application on class instantiation. `timeout` can be specified to change the default timeout used while waiting for an AppleEvent reply.

**TalkTo. **`_start`**()
Test whether the application is running, and attempt to start it if not.

**TalkTo. **`send(code, subcode[, parameters, attributes])`
Create the AppleEvent `Carbon.AE.AEDesc` for the verb with the OSA designation `code, subcode` (which are the usual 4-character strings), pack the `parameters` and `attributes` into it, send it to the target application, wait for the reply, unpack the reply with
unpackevent and return the reply appleevent, the unpacked return values as a dictionary and the return attributes.
38.3. `aepack` — Conversion between Python variables and AppleEvent data containers

The `aepack` module defines functions for converting (packing) Python variables to AppleEvent descriptors and back (unpacking). Within Python the AppleEvent descriptor is handled by Python objects of built-in type `AEDesc`, defined in module `Carbon.AE`.

**Note:** This module has been removed in Python 3.x.

The `aepack` module defines the following functions:

```python
aepack.pack(x[, forcetype])
```

Returns an `AEDesc` object containing a conversion of Python value `x`. If `forcetype` is provided it specifies the descriptor type of the result. Otherwise, a default mapping of Python types to Apple Event descriptor types is used, as follows:

<table>
<thead>
<tr>
<th>Python type</th>
<th>descriptor type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSSpec</td>
<td>typeFSS</td>
</tr>
<tr>
<td>FSRef</td>
<td>typeFSRef</td>
</tr>
<tr>
<td>Alias</td>
<td>typeAlias</td>
</tr>
<tr>
<td>integer</td>
<td>typeLong (32 bit integer)</td>
</tr>
<tr>
<td>float</td>
<td>typeFloat (64 bit floating point)</td>
</tr>
<tr>
<td>string</td>
<td>typeText</td>
</tr>
<tr>
<td>unicode</td>
<td>typeUnicodeText</td>
</tr>
<tr>
<td>list</td>
<td>typeAEList</td>
</tr>
<tr>
<td>dictionary</td>
<td>typeAERecord</td>
</tr>
</tbody>
</table>
If $x$ is a Python instance then this function attempts to call an \texttt{\_\_aepack\_\_()} method. This method should return an \texttt{AEDesc} object.

If the conversion $x$ is not defined above, this function returns the Python string representation of a value (the \texttt{repr()} function) encoded as a text descriptor.

\texttt{aepack.unpack(x[, formodulename])}

$x$ must be an object of type \texttt{AEDesc}. This function returns a Python object representation of the data in the Apple Event descriptor $x$. Simple AppleEvent data types (integer, text, float) are returned as their obvious Python counterparts. Apple Event lists are returned as Python lists, and the list elements are recursively unpacked. Object references (ex. line 3 of document 1) are returned as instances of \texttt{aetypes.ObjectSpecifier}, unless \texttt{formodulename} is specified. AppleEvent descriptors with descriptor type typeFSS are returned as \texttt{FSSpec} objects. AppleEvent record descriptors are returned as Python dictionaries, with 4-character string keys and elements recursively unpacked.

The optional \texttt{formodulename} argument is used by the stub packages generated by \texttt{gensuitemodule}, and ensures that the OSA classes for object specifiers are looked up in the correct module. This ensures that if, say, the Finder returns an object specifier for a window you get an instance of \texttt{Finder.Window} and not a generic \texttt{aetypes.Window}. The former knows about all the properties and elements a window has in the Finder, while the latter knows no such things.

\textbf{See also:}
Module **Carbon.AE**

Built-in access to Apple Event Manager routines.

Module **aetypes**

Python definitions of codes for Apple Event descriptor types.
38.4. `aetypes` — AppleEvent objects

The `aetypes` defines classes used to represent Apple Event data descriptors and Apple Event object specifiers.

Apple Event data is contained in descriptors, and these descriptors are typed. For many descriptors the Python representation is simply the corresponding Python type: `typeText` in OSA is a Python string, `typeFloat` is a float, etc. For OSA types that have no direct Python counterpart this module declares classes. Packing and unpacking instances of these classes is handled automatically by `aepack`.

An object specifier is essentially an address of an object implemented in an Apple Event server. An Apple Event specifier is used as the direct object for an Apple Event or as the argument of an optional parameter. The `aetypes` module contains the base classes for OSA classes and properties, which are used by the packages generated by `gensuitemodule` to populate the classes and properties in a given suite.

For reasons of backward compatibility, and for cases where you need to script an application for which you have not generated the stub package this module also contains object specifiers for a number of common OSA classes such as `Document`, `Window`, `Character`, etc.

Note: This module has been removed in Python 3.x.

The `AEObjects` module defines the following classes to represent Apple Event descriptor data:

```python
class aetypes. Unknown(type, data)
    The representation of OSA descriptor data for which the `aepack`
and aetypes modules have no support, i.e. anything that is not represented by the other classes here and that is not equivalent to a simple Python value.

```python
class aetypes.Enum(enum)
    An enumeration value with the given 4-character string value.

class aetypes.InsertionLoc(of, pos)
    Position pos in object of.

class aetypes.Boolean(bool)
    A boolean.

class aetypes.StyledText(style, text)
    Text with style information (font, face, etc) included.

class aetypes.AEText(script, style, text)
    Text with script system and style information included.

class aetypes.IntlText(script, language, text)
    Text with script system and language information included.

class aetypes.IntlWritingCode(script, language)
    Script system and language information.

class aetypes.QDPoint(v, h)
    A quickdraw point.

class aetypes.QDRectangle(v0, h0, v1, h1)
    A quickdraw rectangle.

class aetypes.RGBColor(r, g, b)
    A color.

class aetypes.Type(type)
    An OSA type value with the given 4-character name.
```
class aetypes. **Keyword**(name)
An OSA keyword with the given 4-character name.

class aetypes. **Range**(start, stop)
A range.

class aetypes. **Ordinal**(abso)
Non-numeric absolute positions, such as "firs", first, or "midd", middle.

class aetypes. **Logical**(logc, term)
The logical expression of applying operator logc to term.

class aetypes. **Comparison**(obj1, relo, obj2)
The comparison relo of obj1 to obj2.

The following classes are used as base classes by the generated stub packages to represent AppleScript classes and properties in Python:

class aetypes. **ComponentItem**(which[, fr])
Abstract baseclass for an OSA class. The subclass should set the class attribute want to the 4-character OSA class code. Instances of subclasses of this class are equivalent to AppleScript Object Specifiers. Upon instantiation you should pass a selector in which, and optionally a parent object in fr.

class aetypes. **NProperty**(fr)
Abstract baseclass for an OSA property. The subclass should set the class attributes want and which to designate which property we are talking about. Instances of subclasses of this class are Object Specifiers.

class aetypes. **ObjectSpecifier**(want, form, seld[, fr])
Base class of `ComponentItem` and `NProperty`, a general OSA Object Specifier. See the Apple Open Scripting Architecture documentation for the parameters. Note that this class is not abstract.
38.5. MiniAEFrame — Open Scripting Architecture server support

The module MiniAEFrame provides a framework for an application that can function as an Open Scripting Architecture (OSA) server, i.e. receive and process AppleEvents. It can be used in conjunction with FrameWork or standalone. As an example, it is used in PythonCGISlave.

The MiniAEFrame module defines the following classes:

**class** MiniAEFrame.**AEServer**  
A class that handles AppleEvent dispatch. Your application should subclass this class together with either MiniApplication or FrameWork.Application. Your __init__() method should call the __init__() method for both classes.

**class** MiniAEFrame.**MiniApplication**  
A class that is more or less compatible with FrameWork.Application but with less functionality. Its event loop supports the apple menu, command-dot and AppleEvents; other events are passed on to the Python interpreter and/or Sioux. Useful if your application wants to use AEServer but does not provide its own windows, etc.
38.5.1. AEServer Objects

AEServer.installahandler(\textit{classe}, \textit{type}, \textit{callback})

Installs an AppleEvent handler. \textit{classe} and \textit{type} are the four-character OSA Class and Type designators, '*****' wildcards are allowed. When a matching AppleEvent is received the parameters are decoded and your callback is invoked.

AEServer.callback(\_\textit{object}, **\textit{kwargs})

Your callback is called with the OSA Direct Object as first positional parameter. The other parameters are passed as keyword arguments, with the 4-character designator as name. Three extra keyword parameters are passed: \_\textit{class} and \_\textit{type} are the Class and Type designators and \_\textit{attributes} is a dictionary with the AppleEvent attributes.

The return value of your method is packed with \texttt{aetools.packevent()} and sent as reply.

Note that there are some serious problems with the current design. AppleEvents which have non-identifier 4-character designators for arguments are not implementable, and it is not possible to return an error to the originator. This will be addressed in a future release.
39. SGI IRIX Specific Services

The modules described in this chapter provide interfaces to features that are unique to SGI's IRIX operating system (versions 4 and 5).

- **39.1. al** — Audio functions on the SGI
  - **39.1.1. Configuration Objects**
  - **39.1.2. Port Objects**
- **39.2. AL** — Constants used with the al module
- **39.3. cd** — CD-ROM access on SGI systems
  - **39.3.1. Player Objects**
  - **39.3.2. Parser Objects**
- **39.4. fl** — FORMS library for graphical user interfaces
  - **39.4.1. Functions Defined in Module fl**
  - **39.4.2. Form Objects**
  - **39.4.3. FORMS Objects**
- **39.5. FL** — Constants used with the fl module
- **39.6. flp** — Functions for loading stored FORMS designs
- **39.7. fm** — *Font Manager* interface
- **39.8. gl** — *Graphics Library* interface
- **39.9. DEVICE** — Constants used with the gl module
- **39.10. GL** — Constants used with the gl module
- **39.11. imgfile** — Support for SGI imglib files
- **39.12. jpeg** — Read and write JPEG files
39.1. **al — Audio functions on the SGI**

*Deprecated since version 2.6:* The `al` module has been removed in Python 3.

This module provides access to the audio facilities of the SGI Indy and Indigo workstations. See section 3A of the IRIX man pages for details. You’ll need to read those man pages to understand what these functions do! Some of the functions are not available in IRIX releases before 4.0.5. Again, see the manual to check whether a specific function is available on your platform.

All functions and methods defined in this module are equivalent to the C functions with `AL` prefixed to their name.

Symbolic constants from the C header file `<audio.h>` are defined in the standard module `AL`, see below.

**Warning:** The current version of the audio library may dump core when bad argument values are passed rather than returning an error status. Unfortunately, since the precise circumstances under which this may happen are undocumented and hard to check, the Python interface can provide no protection against this kind of problems. (One example is specifying an excessive queue size — there is no documented upper limit.)

The module defines the following functions:

```python
al.openport(name, direction[, config])
```

The name and direction arguments are strings. The optional
config argument is a configuration object as returned by newconfig(). The return value is an audio port object; methods of audio port objects are described below.

al. newconfig()
   The return value is a new audio configuration object; methods of audio configuration objects are described below.

al. queryparams(device)
   The device argument is an integer. The return value is a list of integers containing the data returned by ALqueryparams().

al. getparams(device, list)
   The device argument is an integer. The list argument is a list such as returned by queryparams(); it is modified in place (!).

al. setparams(device, list)
   The device argument is an integer. The list argument is a list such as returned by queryparams().
39.1.1. Configuration Objects

Configuration objects returned by `newconfig()` have the following methods:

```python
audio configuration.getqueuesize()
    Return the queue size.

audio configuration.setqueuesize(size)
    Set the queue size.

audio configuration.getwidth()
    Get the sample width.

audio configuration.setwidth(width)
    Set the sample width.

audio configuration.getchannels()
    Get the channel count.

audio configuration.setchannels(nchannels)
    Set the channel count.

audio configuration.getsampfmt()
    Get the sample format.

audio configuration.setsampfmt(sampfmt)
    Set the sample format.

audio configuration.getfloatmax()
    Get the maximum value for floating sample formats.

audio configuration.setfloatmax(floatmax)
    Set the maximum value for floating sample formats.
```
39.1.2. Port Objects

Port objects, as returned by `openport()`, have the following methods:

```python
audio port.closeport()
    Close the port.
```

```python
audio port.getfd()
    Return the file descriptor as an int.
```

```python
audio port.getfilled()
    Return the number of filled samples.
```

```python
audio port.getfillable()
    Return the number of fillable samples.
```

```python
audio port.readsamps(nsamples)
    Read a number of samples from the queue, blocking if necessary. Return the data as a string containing the raw data, (e.g., 2 bytes per sample in big-endian byte order (high byte, low byte) if you have set the sample width to 2 bytes).
```

```python
audio port.writesamps(samples)
    Write samples into the queue, blocking if necessary. The samples are encoded as described for the `readsamps()` return value.
```

```python
audio port.getfillpoint()
    Return the ‘fill point’.
```

```python
audio port.setfillpoint(fillpoint)
    Set the ‘fill point’.
```

```python
audio port.getconfig()
    Return a configuration object containing the current configuration of the port.
```
audio port.setconfig(config)
   Set the configuration from the argument, a configuration object.

audio port.getstatus(list)
   Get status information on last error.
39.2. AL — Constants used with the al module

_Deprecated since version 2.6:_ The _AL_ module has been removed in Python 3.

This module defines symbolic constants needed to use the built-in module _al_ (see above); they are equivalent to those defined in the C header file _<audio.h>_ except that the name prefix _AL_ is omitted. Read the module source for a complete list of the defined names. Suggested use:

```python
import al
from AL import *
```
39.3. cd — CD-ROM access on SGI systems

**Deprecated since version 2.6:** The cd module has been removed in Python 3.

This module provides an interface to the Silicon Graphics CD library. It is available only on Silicon Graphics systems.

The way the library works is as follows. A program opens the CD-ROM device with `open()` and creates a parser to parse the data from the CD with `createparser()`. The object returned by `open()` can be used to read data from the CD, but also to get status information for the CD-ROM device, and to get information about the CD, such as the table of contents. Data from the CD is passed to the parser, which parses the frames, and calls any callback functions that have previously been added.

An audio CD is divided into *tracks* or *programs* (the terms are used interchangeably). Tracks can be subdivided into *indices*. An audio CD contains a *table of contents* which gives the starts of the tracks on the CD. Index 0 is usually the pause before the start of a track. The start of the track as given by the table of contents is normally the start of index 1.

Positions on a CD can be represented in two ways. Either a frame number or a tuple of three values, minutes, seconds and frames. Most functions use the latter representation. Positions can be both relative to the beginning of the CD, and to the beginning of the track.

Module `cd` defines the following functions and constants:
cd. `createparser()`
Create and return an opaque parser object. The methods of the parser object are described below.

cd. `msftoframe(minutes, seconds, frames)`
Converts a `(minutes, seconds, frames)` triple representing time in absolute time code into the corresponding CD frame number.

cd. `open([device[, mode]])`
Open the CD-ROM device. The return value is an opaque player object; methods of the player object are described below. The device is the name of the SCSI device file, e.g. `/dev/scsi/sc0d4l0`, or None. If omitted or None, the hardware inventory is consulted to locate a CD-ROM drive. The mode, if not omitted, should be the string 'r'.

The module defines the following variables:

`exception cd.error`
Exception raised on various errors.

`cd.DATASIZE`
The size of one frame's worth of audio data. This is the size of the audio data as passed to the callback of type audio.

`cd.BLOCKSIZE`
The size of one uninterpreted frame of audio data.

The following variables are states as returned by `getstatus()`:

`cd.READY`
The drive is ready for operation loaded with an audio CD.

`cd.NODISC`
The drive does not have a CD loaded.
The drive is loaded with a CD-ROM. Subsequent play or read operations will return I/O errors.

An error occurred while trying to read the disc or its table of contents.

The drive is in CD player mode playing an audio CD through its audio jacks.

The drive is in CD layer mode with play paused.

The equivalent of PAUSED on older (non 3301) model Toshiba CD-ROM drives. Such drives have never been shipped by SGI.

Integer constants describing the various types of parser callbacks that can be set by the addcallback() method of CD parser objects (see below).
39.3.1. Player Objects

Player objects (returned by `open()`) have the following methods:

**CD player.allowremoval()**
Unlocks the eject button on the CD-ROM drive permitting the user to eject the caddy if desired.

**CD player.bestreadsize()**
Returns the best value to use for the `num_frames` parameter of the `readda()` method. Best is defined as the value that permits a continuous flow of data from the CD-ROM drive.

**CD player.close()**
Frees the resources associated with the player object. After calling `close()`, the methods of the object should no longer be used.

**CD player.eject()**
Ejects the caddy from the CD-ROM drive.

**CD player.getstatus()**
Returns information pertaining to the current state of the CD-ROM drive. The returned information is a tuple with the following values: `state`, `track`, `rtime`, `atime`, `ttme`, `first`, `last`, `scsi_audio`, `cur_block`. `rtime` is the time relative to the start of the current track; `atime` is the time relative to the beginning of the disc; `ttme` is the total time on the disc. For more information on the meaning of the values, see the man page `CDgetstatus(3dm)`. The value of `state` is one of the following: `ERROR`, `NODISC`, `READY`, `PLAYING`, `PAUSED`, `STILL`, or `CDROM`.

**CD player.gettrackinfo(track)**
Returns information about the specified track. The returned
information is a tuple consisting of two elements, the start time of
the track and the duration of the track.

**CD player.msftoblock(min, sec, frame)**
Converts a minutes, seconds, frames triple representing a time in
absolute time code into the corresponding logical block number
for the given CD-ROM drive. You should use `msftoframe()` rather
than `msftoblock()` for comparing times. The logical block number
differs from the frame number by an offset required by certain
CD-ROM drives.

**CD player.play(start, play)**
Starts playback of an audio CD in the CD-ROM drive at the
specified track. The audio output appears on the CD-ROM drive's
headphone and audio jacks (if fitted). Play stops at the end of the
disc. `start` is the number of the track at which to start playing the
CD; if `play` is 0, the CD will be set to an initial paused state. The
method `togglepause()` can then be used to commence play.

**CD player.playabs(minutes, seconds, frames, play)**
Like `play()`, except that the start is given in minutes, seconds,
and frames instead of a track number.

**CD player.playtrack(start, play)**
Like `play()`, except that playing stops at the end of the track.

**CD player.playtrackabs(track, minutes, seconds, frames, play)**
Like `play()`, except that playing begins at the specified absolute
time and ends at the end of the specified track.

**CD player.preventremoval()**
Locks the eject button on the CD-ROM drive thus preventing the
user from arbitrarily ejecting the caddy.

**CD player.readda(num_frames)**
Reads the specified number of frames from an audio CD mounted in the CD-ROM drive. The return value is a string representing the audio frames. This string can be passed unaltered to the parseframe() method of the parser object.

**CD player.seek(minutes, seconds, frames)**
Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to an absolute time code location specified in *minutes*, *seconds*, and *frames*. The return value is the logical block number to which the pointer has been set.

**CD player.seekblock(block)**
Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to the specified logical block number. The return value is the logical block number to which the pointer has been set.

**CD player.seektrack(track)**
Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to the specified track. The return value is the logical block number to which the pointer has been set.

**CD player.stop()**
Stops the current playing operation.

**CD player.togglepause()**
Pauses the CD if it is playing, and makes it play if it is paused.
39.3.2. Parser Objects

Parser objects (returned by `createparser()`) have the following methods:

**CD parser.addcallback(type, func, arg)**

Adds a callback for the parser. The parser has callbacks for eight different types of data in the digital audio data stream. Constants for these types are defined at the `cd` module level (see above). The callback is called as follows: `func(arg, type, data)`, where `arg` is the user supplied argument, `type` is the particular type of callback, and `data` is the data returned for this `type` of callback. The type of the data depends on the `type` of callback as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>audio</td>
<td>String which can be passed unmodified to <code>al.writesamps()</code></td>
</tr>
<tr>
<td>pnum</td>
<td>Integer giving the program (track) number.</td>
</tr>
<tr>
<td>index</td>
<td>Integer giving the index number.</td>
</tr>
<tr>
<td>ptime</td>
<td>Tuple consisting of the program time in minutes, seconds, and frames.</td>
</tr>
<tr>
<td>atime</td>
<td>Tuple consisting of the absolute time in minutes, seconds, and frames.</td>
</tr>
<tr>
<td>catalog</td>
<td>String of 13 characters, giving the catalog number of the CD.</td>
</tr>
<tr>
<td>ident</td>
<td>String of 12 characters, giving the ISRC identification number of the recording. The string consists of two characters country code, three characters owner code, two characters giving the year, and five characters giving a serial number.</td>
</tr>
<tr>
<td>control</td>
<td>Integer giving the control bits from the CD subcode data</td>
</tr>
</tbody>
</table>

**CD parser.deleteparser()**
Deletes the parser and frees the memory it was using. The object should not be used after this call. This call is done automatically when the last reference to the object is removed.

**CD parser.parseframe(frame)**

 Parses one or more frames of digital audio data from a CD such as returned by `readda()`. It determines which subcodes are present in the data. If these subcodes have changed since the last frame, then `parseframe()` executes a callback of the appropriate type passing to it the subcode data found in the frame. Unlike the C function, more than one frame of digital audio data can be passed to this method.

**CD parser.removecallback(type)**

 Removes the callback for the given `type`.

**CD parser.resetparser()**

 Resets the fields of the parser used for tracking subcodes to an initial state. `resetparser()` should be called after the disc has been changed.
39.4. **f1 — FORMS library for graphical user interfaces**

*Deprecated since version 2.6:* The `f1` module has been removed in Python 3.

This module provides an interface to the FORMS Library by Mark Overmars. The source for the library can be retrieved by anonymous FTP from host `ftp.cs.ruu.nl`, directory `SGI/FORMS`. It was last tested with version 2.0b.

Most functions are literal translations of their C equivalents, dropping the initial `f1_` from their name. Constants used by the library are defined in module `FL` described below.

The creation of objects is a little different in Python than in C: instead of the ‘current form’ maintained by the library to which new FORMS objects are added, all functions that add a FORMS object to a form are methods of the Python object representing the form. Consequently, there are no Python equivalents for the C functions `fl_addto_form()` and `fl_end_form()`, and the equivalent of `fl_bgn_form()` is called `f1.make_form()`.

Watch out for the somewhat confusing terminology: FORMS uses the word `object` for the buttons, sliders etc. that you can place in a form. In Python, ‘object’ means any value. The Python interface to FORMS introduces two new Python object types: form objects (representing an entire form) and FORMS objects (representing one button, slider etc.). Hopefully this isn’t too confusing.

There are no ‘free objects’ in the Python interface to FORMS, nor is there an easy way to add object classes written in Python. The
FORMS interface to GL event handling is available, though, so you can mix FORMS with pure GL windows.

Please note: importing f1 implies a call to the GL function foreground() and to the FORMS routine f1_init().
Module \texttt{f1} defines the following functions. For more information about what they do, see the description of the equivalent C function in the FORMS documentation:

\texttt{f1.make_form(type, width, height)}

Create a form with given type, width and height. This returns a \textit{form} object, whose methods are described below.

\texttt{f1.do_forms()}

The standard FORMS main loop. Returns a Python object representing the FORMS object needing interaction, or the special value \texttt{FL.EVENT}.

\texttt{f1.check_forms()}

Check for FORMS events. Returns what \texttt{do_forms()} above returns, or \texttt{None} if there is no event that immediately needs interaction.

\texttt{f1.set_event_call_back(function)}

Set the event callback function.

\texttt{f1.set_graphics_mode(rgbmode, doublebuffering)}

Set the graphics modes.

\texttt{f1.get_rgbmode()}

Return the current rgb mode. This is the value of the C global variable \texttt{fl_rgbmode}.

\texttt{f1.show_message(str1, str2, str3)}

Show a dialog box with a three-line message and an OK button.

\texttt{f1.show_question(str1, str2, str3)}
Show a dialog box with a three-line message and YES and NO buttons. It returns 1 if the user pressed YES, 0 if NO.

```python
fl.show_choice(str1, str2, str3, but1[, but2[, but3]])
```
Show a dialog box with a three-line message and up to three buttons. It returns the number of the button clicked by the user (1, 2 or 3).

```python
fl.show_input(prompt, default)
```
Show a dialog box with a one-line prompt message and text field in which the user can enter a string. The second argument is the default input string. It returns the string value as edited by the user.

```python
fl.show_file_selector(message, directory, pattern, default)
```
Show a dialog box in which the user can select a file. It returns the absolute filename selected by the user, or None if the user presses Cancel.

```python
fl.get_directory()
fl.get_pattern()
fl.get_filename()
```
These functions return the directory, pattern and filename (the tail part only) selected by the user in the last show_file_selector() call.

```python
fl.qdevice(dev)
fl.unqdevice(dev)
fl.isqueued(dev)
fl.qtest()
fl.qread()
fl.qreset()
fl.qenter(dev, val)
fl.get_mouse()
fl.tie(button, valuator1, valuator2)
```
These functions are the FORMS interfaces to the corresponding GL functions. Use these if you want to handle some GL events yourself when using `fl.do_events()`. When a GL event is detected that FORMS cannot handle, `fl.do_forms()` returns the special value `FL.EVENT` and you should call `fl.qread()` to read the event from the queue. Don’t use the equivalent GL functions!

`fl.color()`  
`fl.mapcolor()`  
`fl.getmcolor()`  
See the description in the FORMS documentation of `fl.color()`, `fl.mapcolor()` and `fl.getmcolor()`. 
39.4.2. Form Objects

Form objects (returned by `make_form()` above) have the following methods. Each method corresponds to a C function whose name is prefixed with `fl_`; and whose first argument is a form pointer; please refer to the official FORMS documentation for descriptions.

All the `add_*()` methods return a Python object representing the FORMS object. Methods of FORMS objects are described below. Most kinds of FORMS object also have some methods specific to that kind; these methods are listed here.

```python
form.show_form(placement, bordertype, name)
    Show the form.

form.hide_form()
    Hide the form.

form.redraw_form()
    Redraw the form.

form.set_form_position(x, y)
    Set the form’s position.

form.freeze_form()
    Freeze the form.

form.unfreeze_form()
    Unfreeze the form.

form.activate_form()
    Activate the form.

form.deactivate_form()
```
Deactivate the form.

form. **bgn_group()**
Begin a new group of objects; return a group object.

form. **end_group()**
End the current group of objects.

form. **find_first()**
Find the first object in the form.

form. **find_last()**
Find the last object in the form.

form. **add_box(type, x, y, w, h, name)**
Add a box object to the form. No extra methods.

form. **add_text(type, x, y, w, h, name)**
Add a text object to the form. No extra methods.

form. **add_clock(type, x, y, w, h, name)**
Add a clock object to the form. — Method: **get_clock()**.

form. **add_button(type, x, y, w, h, name)**
Add a button object to the form. — Methods: **get_button()**, **set_button()**.

form. **add_lightbutton(type, x, y, w, h, name)**
Add a lightbutton object to the form. — Methods: **get_button()**, **set_button()**.

form. **add_roundbutton(type, x, y, w, h, name)**
Add a roundbutton object to the form. — Methods: **get_button()**, **set_button()**.

form. **add_slider(type, x, y, w, h, name)**
Add a slider object to the form. — Methods: set_slider_value(), get_slider_value(), set_slider_bounds(), get_slider_bounds(), set_slider_return(), set_slider_size(), set_slider_precision(), set_slider_step().

form.add_valslider(type, x, y, w, h, name)
Add a valslider object to the form. — Methods: set_slider_value(), get_slider_value(), set_slider_bounds(), get_slider_bounds(), set_slider_return(), set_slider_size(), set_slider_precision(), set_slider_step().

form.add_dial(type, x, y, w, h, name)
Add a dial object to the form. — Methods: set_dial_value(), get_dial_value(), set_dial_bounds(), get_dial_bounds().

form.add_positioner(type, x, y, w, h, name)
Add a positioner object to the form. — Methods: set_positioner_xvalue(), set_positioner_yvalue(), set_positioner_xbounds(), set_positioner_ybounds(), get_positioner_xvalue(), get_positioner_yvalue(), get_positioner_xbounds(), get_positioner_ybounds().

form.add_counter(type, x, y, w, h, name)
Add a counter object to the form. — Methods: set_counter_value(), get_counter_value(), set_counter_bounds(), set_counter_step(), set_counter_precision(), set_counter_return().

form.add_input(type, x, y, w, h, name)
Add an input object to the form. — Methods: set_input(), get_input(), set_input_color(), set_input_return().

form.add_menu(type, x, y, w, h, name)
Add a menu object to the form. — Methods: set_menu(), get_menu(), addto_menu().

form. add_choice(type, x, y, w, h, name)
Add a choice object to the form. — Methods: set_choice(), get_choice(), clear_choice(), addto_choice(), replace_choice(), delete_choice(), get_choice_text(), set_choice_fontsize(), set_choice_fontstyle().

form. add_browser(type, x, y, w, h, name)
Add a browser object to the form. — Methods: set_browser_topline(), clear_browser(), add_browser_line(), addto_browser(), insert_browser_line(), delete_browser_line(), replace_browser_line(), get_browser_line(), load_browser(), get_browser_maxline(), select_browser_line(), deselect_browser_line(), deselect_browser(), isselected_browser_line(), get_browser(), set_browser_fontsize(), set_browser_fontstyle(), set_browser_specialkey().

form. add_timer(type, x, y, w, h, name)
Add a timer object to the form. — Methods: set_timer(), get_timer().

Form objects have the following data attributes; see the FORMS documentation:

<table>
<thead>
<tr>
<th>Name</th>
<th>C Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>window</td>
<td>int (read-only)</td>
<td>GL window id</td>
</tr>
<tr>
<td>w</td>
<td>float</td>
<td>form width</td>
</tr>
<tr>
<td>h</td>
<td>float</td>
<td>form height</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
<td>form x origin</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
<td>form y origin</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>-----</td>
<td>----------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>deactivated</strong></td>
<td>int</td>
<td>nonzero if form is deactivated</td>
</tr>
<tr>
<td><strong>visible</strong></td>
<td>int</td>
<td>nonzero if form is visible</td>
</tr>
<tr>
<td><strong>frozen</strong></td>
<td>int</td>
<td>nonzero if form is frozen</td>
</tr>
<tr>
<td><strong>doublebuf</strong></td>
<td>int</td>
<td>nonzero if double buffering on</td>
</tr>
</tbody>
</table>
39.4.3. FORMS Objects

Besides methods specific to particular kinds of FORMS objects, all FORMS objects also have the following methods:

**FORMS object.set_call_back(function, argument)**
Set the object's callback function and argument. When the object needs interaction, the callback function will be called with two arguments: the object, and the callback argument. (FORMS objects without a callback function are returned by `fl.do_forms()` or `fl.check_forms()` when they need interaction.) Call this method without arguments to remove the callback function.

**FORMS object.delete_object()**
Delete the object.

**FORMS object.show_object()**
Show the object.

**FORMS object.hide_object()**
Hide the object.

**FORMS object.redraw_object()**
Redraw the object.

**FORMS object.freeze_object()**
Freeze the object.

**FORMS object.unfreeze_object()**
Unfreeze the object.

FORMS objects have these data attributes; see the FORMS documentation:

<table>
<thead>
<tr>
<th>Name</th>
<th>C Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>objclass</td>
<td>int (read-only)</td>
<td>object class</td>
</tr>
<tr>
<td>type</td>
<td>int (read-only)</td>
<td>object type</td>
</tr>
<tr>
<td>boxtype</td>
<td>int</td>
<td>box type</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
<td>x origin</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
<td>y origin</td>
</tr>
<tr>
<td>w</td>
<td>float</td>
<td>width</td>
</tr>
<tr>
<td>h</td>
<td>float</td>
<td>height</td>
</tr>
<tr>
<td>col1</td>
<td>int</td>
<td>primary color</td>
</tr>
<tr>
<td>col2</td>
<td>int</td>
<td>secondary color</td>
</tr>
<tr>
<td>align</td>
<td>int</td>
<td>alignment</td>
</tr>
<tr>
<td>lcol</td>
<td>int</td>
<td>label color</td>
</tr>
<tr>
<td>lsize</td>
<td>float</td>
<td>label font size</td>
</tr>
<tr>
<td>label</td>
<td>string</td>
<td>label string</td>
</tr>
<tr>
<td>lstyle</td>
<td>int</td>
<td>label style</td>
</tr>
<tr>
<td>pushed</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>focus</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>belowmouse</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>frozen</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>active</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>input</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>visible</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>radio</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>automatic</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
</tbody>
</table>
39.5. **FL — Constants used with the f1 module**

*Deprecated since version 2.6:* The `FL` module has been removed in Python 3.

This module defines symbolic constants needed to use the built-in module `f1` (see above); they are equivalent to those defined in the C header file `<forms.h>` except that the name prefix `FL_` is omitted. Read the module source for a complete list of the defined names. Suggested use:

```python
import f1
from FL import *
```
39.6. \texttt{flp} — Functions for loading stored FORMS designs

\textit{Deprecated since version 2.6:} The \texttt{flp} module has been removed in Python 3.

This module defines functions that can read form definitions created by the ‘form designer’ (\texttt{fdesign}) program that comes with the FORMS library (see module \texttt{f1} above).

For now, see the file \texttt{flp.doc} in the Python library source directory for a description.

XXX A complete description should be inserted here!
39.7. fm — Font Manager interface

Deprecation notice: The fm module has been removed in Python 3.

This module provides access to the IRIS Font Manager library. It is available only on Silicon Graphics machines. See also: 4Sight User's Guide, section 1, chapter 5: “Using the IRIS Font Manager.”

This is not yet a full interface to the IRIS Font Manager. Among the unsupported features are: matrix operations; cache operations; character operations (use string operations instead); some details of font info; individual glyph metrics; and printer matching.

It supports the following operations:

- **fm.init()**
  Initialization function. Calls fminit(). It is normally not necessary to call this function, since it is called automatically the first time the fm module is imported.

- **fm.findfont(fontname)**
  Return a font handle object. Calls fmfindfont(fontname).

- **fm.enumerate()**
  Returns a list of available font names. This is an interface to fmenumerate().

- **fm.prstr(string)**
  Render a string using the current font (see the setfont() font handle method below). Calls fmprstr(string).

- **fm.setpath(string)**
Sets the font search path. Calls fmsetpath(string). (XXX Does not work!?!)

```
fm.fontpath()
```

Returns the current font search path.

Font handle objects support the following operations:

```
font handle.scalefont(factor)
```

Returns a handle for a scaled version of this font. Calls fmscalefont(fh, factor).

```
font handle.setfont()
```

Makes this font the current font. Note: the effect is undone silently when the font handle object is deleted. Calls fmsetfont(fh).

```
font handle.getfontname()
```

Returns this font's name. Calls fmgetfontname(fh).

```
font handle.getcomment()
```

Returns the comment string associated with this font. Raises an exception if there is none. Calls fmgetcomment(fh).

```
font handle.getfontinfo()
```

Returns a tuple giving some pertinent data about this font. This is an interface to fmgetfontinfo(). The returned tuple contains the following numbers: (printermatched, fixed_width, xorig, yorig, xsize, ysize, height, nglyphs).

```
font handle.getstrwidth(string)
```

Returns the width, in pixels, of string when drawn in this font. Calls fmgetstrwidth(fh, string).
39.8. g1 — Graphics Library interface

*Deprecated since version 2.6:* The g1 module has been removed in Python 3.

This module provides access to the Silicon Graphics Graphics Library. It is available only on Silicon Graphics machines.

**Warning:** Some illegal calls to the GL library cause the Python interpreter to dump core. In particular, the use of most GL calls is unsafe before the first window is opened.

The module is too large to document here in its entirety, but the following should help you to get started. The parameter conventions for the C functions are translated to Python as follows:

- All (short, long, unsigned) int values are represented by Python integers.

- All float and double values are represented by Python floating point numbers. In most cases, Python integers are also allowed.

- All arrays are represented by one-dimensional Python lists. In most cases, tuples are also allowed.

- All string and character arguments are represented by Python strings, for instance, `winopen('Hi There!')` and `rotate(900, 'z')`.

- All (short, long, unsigned) integer arguments or return values that are only used to specify the length of an array argument are
omitted. For example, the C call

```c
lmdef(deftype, index, np, props)
```

is translated to Python as

```python
lmdef(deftype, index, props)
```

- Output arguments are omitted from the argument list; they are transmitted as function return values instead. If more than one value must be returned, the return value is a tuple. If the C function has both a regular return value (that is not omitted because of the previous rule) and an output argument, the return value comes first in the tuple. Examples: the C call

```c
getmcolor(i, &red, &green, &blue)
```

is translated to Python as

```python
red, green, blue = getmcolor(i)
```

The following functions are non-standard or have special argument conventions:

- **gl. varray(argument)**
  Equivalent to but faster than a number of `v3d()` calls. The argument is a list (or tuple) of points. Each point must be a tuple of coordinates `(x, y, z)` or `(x, y)`. The points may be 2- or 3-dimensional but must all have the same dimension. Float and int values may be mixed however. The points are always converted to 3D double precision points by assuming `z = 0.0` if necessary (as indicated in the man page), and for each point `v3d()` is called.

- **gl. nvarray()**
  Equivalent to but faster than a number of `n3f` and `v3f` calls. The
argument is an array (list or tuple) of pairs of normals and points. Each pair is a tuple of a point and a normal for that point. Each point or normal must be a tuple of coordinates \((x, y, z)\). Three coordinates must be given. Float and int values may be mixed. For each pair, \(n3f()\) is called for the normal, and then \(v3f()\) is called for the point.

\[\text{gl.} \ vnarray()\]
Similar to \(nvarray()\) but the pairs have the point first and the normal second.

\[\text{gl.} \ nurbssurface(s_k, t_k, ctl, s_ord, t_ord, type)\]
Defines a nurbs surface. The dimensions of \(ctl[][]\) are computed as follows: \([\text{len}(s_k) - s_{ord}], [\text{len}(t_k) - t_{ord}]\).

\[\text{gl.} \ nurbscurve(knots, ctlpoints, order, type)\]
Defines a nurbs curve. The length of \(ctlpoints\) is \(\text{len}(knots) - order\).

\[\text{gl.} \ pw1curve(points, type)\]
Defines a piecewise-linear curve. \(points\) is a list of points. \(type\) must be \(N\_ST\).

\[\text{gl.} \ pick(n)\]
\[\text{gl.} \ select(n)\]
The only argument to these functions specifies the desired size of the pick or select buffer.

\[\text{gl.} \ endpick()\]
\[\text{gl.} \ endselect()\]
These functions have no arguments. They return a list of integers representing the used part of the pick/select buffer. No method is provided to detect buffer overrun.

Here is a tiny but complete example GL program in Python:
import gl, GL, time

def main():
    gl.foreground()
    gl.prefposition(500, 900, 500, 900)
    w = gl.winopen('CrissCross')
    gl.ortho2(0.0, 400.0, 0.0, 400.0)
    gl.color(GL.WHITE)
    gl.clear()
    gl.color(GL.RED)
    gl.bgnline()
    gl.v2f(0.0, 0.0)
    gl.v2f(400.0, 0.0)
    gl.endline()
    gl.bgnline()
    gl.v2f(0.0, 0.0)
    gl.v2f(400.0, 400.0)
    gl.endline()
    time.sleep(5)

See also:

PyOpenGL: The Python OpenGL Binding

An interface to OpenGL is also available; see information about the PyOpenGL project online at http://pyopengl.sourceforge.net/. This may be a better option if support for SGI hardware from before about 1996 is not required.
39.9. **DEVICE** — Constants used with the gl module

*Deprecated since version 2.6:* The **DEVICE** module has been removed in Python 3.

This module defines the constants used by the Silicon Graphics Graphics Library that C programmers find in the header file `<gl/device.h>`. Read the module source file for details.
39.10. GL — Constants used with the gl module

Deprecation warning: The `GL` module has been removed in Python 3.

This module contains constants used by the Silicon Graphics Graphics Library from the C header file `<gl/gl.h>`. Read the module source file for details.
39.11. imgfile — Support for SGI imglib files

*Deprecated since version 2.6:* The `imgfile` module has been removed in Python 3.

The `imgfile` module allows Python programs to access SGI imglib image files (also known as `.rgb` files). The module is far from complete, but is provided anyway since the functionality that there is enough in some cases. Currently, colormap files are not supported.

The module defines the following variables and functions:

- **exception** `imgfile.error`  
  This exception is raised on all errors, such as unsupported file type, etc.

- **`imgfile.getsizes(file)`**  
  This function returns a tuple `(x, y, z)` where `x` and `y` are the size of the image in pixels and `z` is the number of bytes per pixel. Only 3 byte RGB pixels and 1 byte greyscale pixels are currently supported.

- **`imgfile.read(file)`**  
  This function reads and decodes the image on the specified file, and returns it as a Python string. The string has either 1 byte greyscale pixels or 4 byte RGBA pixels. The bottom left pixel is the first in the string. This format is suitable to pass to `gl.lrectwrite()`, for instance.

- **`imgfile.readscaled(file, x, y, filter[, blur])`**  
  This function is identical to `read` but it returns an image that is
scaled to the given $x$ and $y$ sizes. If the \texttt{filter} and \texttt{blur} parameters are omitted scaling is done by simply dropping or duplicating pixels, so the result will be less than perfect, especially for computer-generated images.

Alternatively, you can specify a filter to use to smooth the image after scaling. The filter forms supported are \texttt{impulse'}, \texttt{box'}, \texttt{triangle'}, \texttt{quadratic} and \texttt{gaussian}. If a filter is specified \texttt{blur} is an optional parameter specifying the blurriness of the filter. It defaults to 1.0.

\texttt{readscaled()} makes no attempt to keep the aspect ratio correct, so that is the users' responsibility.

\texttt{imgfile.ttob(flag)}

This function sets a global flag which defines whether the scan lines of the image are read or written from bottom to top (flag is zero, compatible with SGI GL) or from top to bottom (flag is one, compatible with X). The default is zero.

\texttt{imgfile.write(file, data, x, y, z)}

This function writes the RGB or greyscale data in \texttt{data} to image file \texttt{file}. \texttt{x} and \texttt{y} give the size of the image, \texttt{z} is 1 for 1 byte greyscale images or 3 for RGB images (which are stored as 4 byte values of which only the lower three bytes are used). These are the formats returned by \texttt{gl.lrectread()}. 
39.12. jpeg — Read and write JPEG files

*Deprecated since version 2.6: The jpeg module has been removed in Python 3.*

The module `jpeg` provides access to the jpeg compressor and decompressor written by the Independent JPEG Group (IJG). JPEG is a standard for compressing pictures; it is defined in ISO 10918. For details on JPEG or the Independent JPEG Group software refer to the JPEG standard or the documentation provided with the software.

A portable interface to JPEG image files is available with the Python Imaging Library (PIL) by Fredrik Lundh. Information on PIL is available at [http://www.pythonware.com/products/pil/](http://www.pythonware.com/products/pil/).

The `jpeg` module defines an exception and some functions.

*exception* `jpeg.error`

Exception raised by `compress()` and `decompress()` in case of errors.

`jpeg.compress(data, w, h, b)`

Treat data as a pixmap of width `w` and height `h`, with `b` bytes per pixel. The data is in SGI GL order, so the first pixel is in the lower-left corner. This means that `gl.lrectread()` return data can immediately be passed to `compress()`. Currently only 1 byte and 4 byte pixels are allowed, the former being treated as greyscale and the latter as RGB color. `compress()` returns a string that contains the compressed picture, in JFIF format.
**jpeg.decompress(data)**

Data is a string containing a picture in JFIF format. It returns a tuple `(data, width, height, bytesperpixel)`. Again, the data is suitable to pass to `gl.lrectwrite()`.

**jpeg.setoption(name, value)**

Set various options. Subsequent `compress()` and `decompress()` calls will use these options. The following options are available:

<table>
<thead>
<tr>
<th>Option</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>'forcegray'</td>
<td>Force output to be grayscale, even if input is RGB.</td>
</tr>
<tr>
<td>'quality'</td>
<td>Set the quality of the compressed image to a value between 0 and 100 (default is 75). This only affects compression.</td>
</tr>
<tr>
<td>'optimize'</td>
<td>Perform Huffman table optimization. Takes longer, but results in smaller compressed image. This only affects compression.</td>
</tr>
<tr>
<td>'smooth'</td>
<td>Perform inter-block smoothing on uncompressed image. Only useful for low-quality images. This only affects decompression.</td>
</tr>
</tbody>
</table>

**See also:**

**JPEG Still Image Data Compression Standard**

The canonical reference for the JPEG image format, by Pennebaker and Mitchell.

**Information Technology - Digital Compression and Coding of Continuous-tone Still Images - Requirements and Guidelines**

The ISO standard for JPEG is also published as ITU T.81. This is available online in PDF form.
40. SunOS Specific Services

The modules described in this chapter provide interfaces to features that are unique to SunOS 5 (also known as Solaris version 2).

- 40.1. sunaudiodev — Access to Sun audio hardware
  - 40.1.1. Audio Device Objects
- 40.2. SUNAUDIODEV — Constants used with sunaudiodev
40.1. sunaudiodev — Access to Sun audio hardware

*Deprecated since version 2.6:* The `sunaudiodev` module has been removed in Python 3.

This module allows you to access the Sun audio interface. The Sun audio hardware is capable of recording and playing back audio data in u-LAW format with a sample rate of 8K per second. A full description can be found in the `audio(7I)` manual page.

The module `SUNAUDIODEV` defines constants which may be used with this module.

This module defines the following variables and functions:

* `exception sunaudiodev.error`
  This exception is raised on all errors. The argument is a string describing what went wrong.

* `sunaudiodev.open(mode)`
  This function opens the audio device and returns a Sun audio device object. This object can then be used to do I/O on. The `mode` parameter is one of 'r' for record-only access, 'w' for play-only access, 'rw' for both and 'control' for access to the control device. Since only one process is allowed to have the recorder or player open at the same time it is a good idea to open the device only for the activity needed. See `audio(7I)` for details.

As per the manpage, this module first looks in the environment variable `AUDIODEV` for the base audio device filename. If not found, it falls back to `/dev/audio`. The control device is calculated
by appending “ctl” to the base audio device.
40.1.1. Audio Device Objects

The audio device objects are returned by `open()` define the following methods (except `control` objects which only provide `getinfo()`, `setinfo()`, `fileno()`, and `drain()`):

**audio device.close()**
This method explicitly closes the device. It is useful in situations where deleting the object does not immediately close it since there are other references to it. A closed device should not be used again.

**audio device.fileno()**
Returns the file descriptor associated with the device. This can be used to set up `SIGPOLL` notification, as described below.

**audio device.drain()**
This method waits until all pending output is processed and then returns. Calling this method is often not necessary: destroying the object will automatically close the audio device and this will do an implicit drain.

**audio device.flush()**
This method discards all pending output. It can be used avoid the slow response to a user’s stop request (due to buffering of up to one second of sound).

**audio device.getinfo()**
This method retrieves status information like input and output volume, etc. and returns it in the form of an audio status object. This object has no methods but it contains a number of attributes describing the current device status. The names and meanings of the attributes are described in `<sun/audioioio.h>` and in the `audio(7I)` manual page. Member names are slightly different from
their C counterparts: a status object is only a single structure. Members of the play substructure have o_ prepended to their name and members of the record structure have i_. So, the C member play.sample_rate is accessed as o_sample_rate, record.gain as i_gain and monitor_gain plainly as monitor_gain.

audio device.ibufcount()
This method returns the number of samples that are buffered on the recording side, i.e. the program will not block on a read() call of so many samples.

audio device.obufcount()
This method returns the number of samples buffered on the playback side. Unfortunately, this number cannot be used to determine a number of samples that can be written without blocking since the kernel output queue length seems to be variable.

audio device.read(size)
This method reads size samples from the audio input and returns them as a Python string. The function blocks until enough data is available.

audio device.setinfo(status)
This method sets the audio device status parameters. The status parameter is a device status object as returned by getinfo() and possibly modified by the program.

audio device.write(samples)
Write is passed a Python string containing audio samples to be played. If there is enough buffer space free it will immediately return, otherwise it will block.

The audio device supports asynchronous notification of various events, through the SIGPOLL signal. Here’s an example of how you
might enable this in Python:

```python
def handle_sigpoll(signum, frame):
    print 'I got a SIGPOLL update'

import fcntl, signal, STROPTS

signal.signal(signal.SIGPOLL, handle_sigpoll)
fcntl.ioctl(audio_obj.fileno(), STROPTS.I_SETSIG, STROPTS.S_MSG)
```
40.2. **SUNAUDIODEV** — Constants used with **sunaudiodev**

*Deprecated since version 2.6:* The **SUNAUDIODEV** module has been removed in Python 3.

This is a companion module to **sunaudiodev** which defines useful symbolic constants like **MIN_GAIN**, **MAX_GAIN**, **SPEAKER**, etc. The names of the constants are the same names as used in the C include file `<sun/audioio.h>`, with the leading string **AUDIO** stripped.
41. Undocumented Modules

Here’s a quick listing of modules that are currently undocumented, but that should be documented. Feel free to contribute documentation for them! (Send via email to docs@python.org.)

The idea and original contents for this chapter were taken from a posting by Fredrik Lundh; the specific contents of this chapter have been substantially revised.
41.1. Miscellaneous useful utilities

Some of these are very old and/or not very robust; marked with “hmm.”

ihooks
— Import hook support (for rexec; may become obsolete). Removed in Python 3.x.
41.2. Platform specific modules

These modules are used to implement the `os.path` module, and are not documented beyond this mention. There's little need to document these.

**ntpath**
- Implementation of `os.path` on Win32, Win64, WinCE, and OS/2 platforms.

**posixpath**
- Implementation of `os.path` on POSIX.

**bsddb185**
- Backwards compatibility module for systems which still use the Berkeley DB 1.85 module. It is normally only available on certain BSD Unix-based systems. It should never be used directly.
41.3. Multimedia

**audiodev**
— Platform-independent API for playing audio data. Removed in Python 3.x.

**linuxaudiodev**
— Play audio data on the Linux audio device. Replaced in Python 2.3 by the **ossaudiodev** module. Removed in Python 3.x.

**sunaudio**
— Interpret Sun audio headers (may become obsolete or a tool/demo). Removed in Python 3.x.

**toaiff**
— Convert “arbitrary” sound files to AIFF files; should probably become a tool or demo. Requires the external program **sox**. Removed in Python 3.x.
41.4. Undocumented Mac OS modules

41.4.1. applesingle — AppleSingle decoder

*Deprecated since version 2.6.*

41.4.2. buildtools — Helper module for BuildApplet and Friends

*Deprecated since version 2.4.*

41.4.3. cfmyfile — Code Fragment Resource module

*cfmyfile* is a module that understands Code Fragments and the accompanying “cfrg” resources. It can parse them and merge them, and is used by BuildApplication to combine all plugin modules to a single executable.

*Deprecated since version 2.4.*

41.4.4. icopen — Internet Config replacement for open()

Importing *icopen* will replace the built-in *open()* with a version that uses Internet Config to set file type and creator for new files.
41.4.5. `macerrors` — Mac OS Errors

`macerrors` contains constant definitions for many Mac OS error codes.

41.4.6. `macresource` — Locate script resources

`macresource` helps scripts finding their resources, such as dialogs and menus, without requiring special case code for when the script is run under MacPython, as a MacPython applet or under OSX Python.

41.4.7. `Nav` — NavServices calls

A low-level interface to Navigation Services.

41.4.8. `PixMapWrapper` — Wrapper for PixMap objects

`PixMapWrapper` wraps a PixMap object with a Python object that allows access to the fields by name. It also has methods to convert
to and from PIL images.

* Deprecated since version 2.6.*

### 41.4.9. videoreader — Read QuickTime movies

*videoreader* reads and decodes QuickTime movies and passes a stream of images to your program. It also provides some support for audio tracks.

* Deprecated since version 2.6.*

### 41.4.10. w — Widgets built on Framework

The *w* widgets are used extensively in the IDE.

* Deprecated since version 2.6.*
41.5. Obsolete

These modules are not normally available for import; additional work must be done to make them available.

These extension modules written in C are not built by default. Under Unix, these must be enabled by uncommenting the appropriate lines in Modules/Setup in the build tree and either rebuilding Python if the modules are statically linked, or building and installing the shared object if using dynamically-loaded extensions.

**timing**

— Measure time intervals to high resolution (use `time.clock()` instead). Removed in Python 3.x.
41.6. SGI-specific Extension modules

The following are SGI specific, and may be out of touch with the current version of reality.

c1
— Interface to the SGI compression library.

sv
— Interface to the “simple video” board on SGI Indigo (obsolete hardware). Removed in Python 3.x.
1. Extending Python with C or C++

It is quite easy to add new built-in modules to Python, if you know how to program in C. Such extension modules can do two things that can’t be done directly in Python: they can implement new built-in object types, and they can call C library functions and system calls.

To support extensions, the Python API (Application Programmers Interface) defines a set of functions, macros and variables that provide access to most aspects of the Python run-time system. The Python API is incorporated in a C source file by including the header "Python.h".

The compilation of an extension module depends on its intended use as well as on your system setup; details are given in later chapters.

**Note:** The C extension interface is specific to CPython, and extension modules do not work on other Python implementations. In many cases, it is possible to avoid writing C extensions and preserve portability to other implementations. For example, if your use case is calling C library functions or system calls, you should consider using the `ctypes` module or the `cffi` library rather than writing custom C code. These modules let you write Python code to interface with C code and are more portable between implementations of Python than writing and compiling a C extension module.
1.1. A Simple Example

Let’s create an extension module called spam (the favorite food of Monty Python fans...) and let’s say we want to create a Python interface to the C library function system(). [1] This function takes a null-terminated character string as argument and returns an integer. We want this function to be callable from Python as follows:

```python
>>> import spam
>>> status = spam.system("ls -l")
```

Begin by creating a file spammodule.c. (Historically, if a module is called spam, the C file containing its implementation is called spammodule.c; if the module name is very long, like spammify, the module name can be just spammify.c.)

The first line of our file can be:

```c
#include <Python.h>
```

which pulls in the Python API (you can add a comment describing the purpose of the module and a copyright notice if you like).

**Note:** Since Python may define some pre-processor definitions which affect the standard headers on some systems, you *must* include Python.h before any standard headers are included.

All user-visible symbols defined by Python.h have a prefix of Py or PY, except those defined in standard header files. For convenience, and since they are used extensively by the Python interpreter, "Python.h" includes a few standard header files: <stdio.h>, <string.h>, <errno.h>, and <stdlib.h>. If the latter header file does
not exist on your system, it declares the functions `malloc()`, `free()` and `realloc()` directly.

The next thing we add to our module file is the C function that will be called when the Python expression `spam.system(string)` is evaluated (we’ll see shortly how it ends up being called):

```c
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return Py_BuildValue("i", sts);
}
```

There is a straightforward translation from the argument list in Python (for example, the single expression "ls -l") to the arguments passed to the C function. The C function always has two arguments, conventionally named `self` and `args`.

For module functions, the `self` argument is `NULL` or a pointer selected while initializing the module (see `Py_InitModule4()`). For a method, it would point to the object instance.

The `args` argument will be a pointer to a Python tuple object containing the arguments. Each item of the tuple corresponds to an argument in the call’s argument list. The arguments are Python objects — in order to do anything with them in our C function we have to convert them to C values. The function `PyArg_ParseTuple()` in the Python API checks the argument types and converts them to C values. It uses a template string to determine the required types of the arguments as well as the types of the C variables into which to store the converted values. More about this later.
PyArg_ParseTuple() returns true (nonzero) if all arguments have the right type and its components have been stored in the variables whose addresses are passed. It returns false (zero) if an invalid argument list was passed. In the latter case it also raises an appropriate exception so the calling function can return NULL immediately (as we saw in the example).
1.2. Intermezzo: Errors and Exceptions

An important convention throughout the Python interpreter is the following: when a function fails, it should set an exception condition and return an error value (usually a NULL pointer). Exceptions are stored in a static global variable inside the interpreter; if this variable is NULL no exception has occurred. A second global variable stores the “associated value” of the exception (the second argument to raise). A third variable contains the stack traceback in case the error originated in Python code. These three variables are the C equivalents of the Python variables `sys.exc_type`, `sys.exc_value` and `sys.exc_traceback` (see the section on module `sys` in the Python Library Reference). It is important to know about them to understand how errors are passed around.

The Python API defines a number of functions to set various types of exceptions.

The most common one is `PyErr_SetString()`. Its arguments are an exception object and a C string. The exception object is usually a predefined object like `PyExc_ZeroDivisionError`. The C string indicates the cause of the error and is converted to a Python string object and stored as the “associated value” of the exception.

Another useful function is `PyErr_SetFromErrno()`, which only takes an exception argument and constructs the associated value by inspection of the global variable `errno`. The most general function is `PyErr_SetObject()`, which takes two object arguments, the exception and its associated value. You don’t need to `Py_INCREF()` the objects passed to any of these functions.

You can test non-destructively whether an exception has been set with `PyErr_Occurred()`. This returns the current exception object, or
NULL if no exception has occurred. You normally don’t need to call
PyErr_Occurred() to see whether an error occurred in a function call,
since you should be able to tell from the return value.

When a function $f$ that calls another function $g$ detects that the latter
fails, $f$ should itself return an error value (usually NULL or -1). It
should not call one of the PyErr_*() functions — one has already
been called by $g$. $f$’s caller is then supposed to also return an error
indication to its caller, again without calling PyErr_*(), and so on —
the most detailed cause of the error was already reported by the
function that first detected it. Once the error reaches the Python
interpreter’s main loop, this aborts the currently executing Python
code and tries to find an exception handler specified by the Python
programmer.

(There are situations where a module can actually give a more
detailed error message by calling another PyErr_*() function, and in
such cases it is fine to do so. As a general rule, however, this is not
necessary, and can cause information about the cause of the error to
be lost: most operations can fail for a variety of reasons.)

To ignore an exception set by a function call that failed, the exception
condition must be cleared explicitly by calling PyErr_Clear(). The
only time C code should call PyErr_Clear() is if it doesn’t want to
pass the error on to the interpreter but wants to handle it completely
by itself (possibly by trying something else, or pretending nothing
went wrong).

Every failing malloc() call must be turned into an exception — the
direct caller of malloc() (or realloc()) must call PyErr_NoMemory()
and return a failure indicator itself. All the object-creating functions
(for example, PyInt_FromLong()) already do this, so this note is only
relevant to those who call malloc() directly.
Also note that, with the important exception of `PyArg_ParseTuple()` and friends, functions that return an integer status usually return a positive value or zero for success and -1 for failure, like Unix system calls.

Finally, be careful to clean up garbage (by making `Py_XDECREF()` or `Py_DECREF()` calls for objects you have already created) when you return an error indicator!

The choice of which exception to raise is entirely yours. There are predeclared C objects corresponding to all built-in Python exceptions, such as `PyExc_ZeroDivisionError`, which you can use directly. Of course, you should choose exceptions wisely — don’t use `PyExc_TypeError` to mean that a file couldn’t be opened (that should probably be `PyExc_IOError`). If something’s wrong with the argument list, the `PyArg_ParseTuple()` function usually raises `PyExc_TypeError`. If you have an argument whose value must be in a particular range or must satisfy other conditions, `PyExc_ValueError` is appropriate.

You can also define a new exception that is unique to your module. For this, you usually declare a static object variable at the beginning of your file:

```c
static PyObject *SpamError;
```

and initialize it in your module’s initialization function (`initspam()`) with an exception object (leaving out the error checking for now):

```c
PyMODINIT_FUNC
initspam(void)
{
    PyObject *m;

    m = Py_InitModule("spam", SpamMethods);
    if (m == NULL)
```
Note that the Python name for the exception object is `spam.error`. The `PyErr_NewException()` function may create a class with the base class being `Exception` (unless another class is passed in instead of `NULL`), described in Built-in Exceptions.

Note also that the `SpamError` variable retains a reference to the newly created exception class; this is intentional! Since the exception could be removed from the module by external code, an owned reference to the class is needed to ensure that it will not be discarded, causing `SpamError` to become a dangling pointer. Should it become a dangling pointer, C code which raises the exception could cause a core dump or other unintended side effects.

We discuss the use of `PyMODINIT_FUNC` as a function return type later in this sample.

The `spam.error` exception can be raised in your extension module using a call to `PyErr_SetString()` as shown below:

```c
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    if (sts < 0) {
        PyErr_SetString(SpamError, "System command failed");
        return NULL;
    }
```
return PyLong_FromLong(sts);
}
1.3. Back to the Example

Going back to our example function, you should now be able to understand this statement:

```c
if (!PyArg_ParseTuple(args, "s", &command))
    return NULL;
```

It returns `NULL` (the error indicator for functions returning object pointers) if an error is detected in the argument list, relying on the exception set by `PyArg_ParseTuple()`. Otherwise the string value of the argument has been copied to the local variable `command`. This is a pointer assignment and you are not supposed to modify the string to which it points (so in Standard C, the variable `command` should properly be declared as `const char *command`).

The next statement is a call to the Unix function `system()`, passing it the string we just got from `PyArg_ParseTuple()`:

```c
sts = system(command);
```

Our `spam.system()` function must return the value of `sts` as a Python object. This is done using the function `Py_BuildValue()`, which is something like the inverse of `PyArg_ParseTuple()`: it takes a format string and an arbitrary number of C values, and returns a new Python object. More info on `Py_BuildValue()` is given later.

```c
return Py_BuildValue("i", sts);
```

In this case, it will return an integer object. (Yes, even integers are objects on the heap in Python!)

If you have a C function that returns no useful argument (a function
returning `void`), the corresponding Python function must return `None`. You need this idiom to do so (which is implemented by the `Py_RETURN_NONE` macro):

```c
Py_INCREF(Py_None);
return Py_None;
```

`Py_None` is the C name for the special Python object `None`. It is a genuine Python object rather than a `NULL` pointer, which means “error” in most contexts, as we have seen.
1.4. The Module’s Method Table and Initialization Function

I promised to show how `spam_system()` is called from Python programs. First, we need to list its name and address in a “method table”:

```c
static PyMethodDef SpamMethods[] = {
    ...,
    {"system", spam_system, METH_VARARGS, "Execute a shell command."},
    ...,
    {NULL, NULL, 0, NULL}, /* Sentinel */
};
```

Note the third entry (METH_VARARGS). This is a flag telling the interpreter the calling convention to be used for the C function. It should normally always be METH_VARARGS or METH_VARARGS | METH_KEYWORDS; a value of 0 means that an obsolete variant of `PyArg_ParseTuple()` is used.

When using only METH_VARARGS, the function should expect the Python-level parameters to be passed in as a tuple acceptable for parsing via `PyArg_ParseTuple()`; more information on this function is provided below.

The METH_KEYWORDS bit may be set in the third field if keyword arguments should be passed to the function. In this case, the C function should accept a third `PyObject *` parameter which will be a dictionary of keywords. Use `PyArg_ParseTupleAndKeywords()` to parse the arguments to such a function.

The method table must be passed to the interpreter in the module’s initialization function. The initialization function must be named
initname(), where name is the name of the module, and should be the only non-static item defined in the module file:

```c
PyMODINIT_FUNC
initspam(void)
{
    (void) Py_InitModule("spam", SpamMethods);
}
```

Note that PyMODINIT_FUNC declares the function as void return type, declares any special linkage declarations required by the platform, and for C++ declares the function as extern "C".

When the Python program imports module spam for the first time, initspam() is called. (See below for comments about embedding Python.) It calls Py_InitModule(), which creates a “module object” (which is inserted in the dictionary sys.modules under the key "spam"), and inserts built-in function objects into the newly created module based upon the table (an array of PyMethodDef structures) that was passed as its second argument. Py_InitModule() returns a pointer to the module object that it creates (which is unused here). It may abort with a fatal error for certain errors, or return NULL if the module could not be initialized satisfactorily.

When embedding Python, the initspam() function is not called automatically unless there's an entry in the _PyImport_Inittab table. The easiest way to handle this is to statically initialize your statically-linked modules by directly calling initspam() after the call to Py_Initialize():

```c
int
main(int argc, char *argv[])
{
    /* Pass argv[0] to the Python interpreter */
    Py_SetProgramName(argv[0]);
```
/* Initialize the Python interpreter. Required. */
Py_Initialize();

/* Add a static module */
initspam();
...

An example may be found in the file `Demo/embed/demo.c` in the Python source distribution.

**Note:** Removing entries from `sys.modules` or importing compiled modules into multiple interpreters within a process (or following a `fork()` without an intervening `exec()`) can create problems for some extension modules. Extension module authors should exercise caution when initializing internal data structures. Note also that the `reload()` function can be used with extension modules, and will call the module initialization function (`initspam()` in the example), but will not load the module again if it was loaded from a dynamically loadable object file (.so on Unix, .dll on Windows).

A more substantial example module is included in the Python source distribution as `Modules/xxmodule.c`. This file may be used as a template or simply read as an example.
1.5. Compilation and Linkage

There are two more things to do before you can use your new extension: compiling and linking it with the Python system. If you use dynamic loading, the details may depend on the style of dynamic loading your system uses; see the chapters about building extension modules (chapter Building C and C++ Extensions with distutils) and additional information that pertains only to building on Windows (chapter Building C and C++ Extensions on Windows) for more information about this.

If you can’t use dynamic loading, or if you want to make your module a permanent part of the Python interpreter, you will have to change the configuration setup and rebuild the interpreter. Luckily, this is very simple on Unix: just place your file (spammodule.c for example) in the Modules/ directory of an unpacked source distribution, add a line to the file Modules/Setup.local describing your file:

    spam spammodule.o

and rebuild the interpreter by running make in the toplevel directory. You can also run make in the Modules/ subdirectory, but then you must first rebuild Makefile there by running ‘make Makefile’. (This is necessary each time you change the Setup file.)

If your module requires additional libraries to link with, these can be listed on the line in the configuration file as well, for instance:

    spam spammodule.o -lx11
1.6. Calling Python Functions from C

So far we have concentrated on making C functions callable from Python. The reverse is also useful: calling Python functions from C. This is especially the case for libraries that support so-called “callback” functions. If a C interface makes use of callbacks, the equivalent Python often needs to provide a callback mechanism to the Python programmer; the implementation will require calling the Python callback functions from a C callback. Other uses are also imaginable.

Fortunately, the Python interpreter is easily called recursively, and there is a standard interface to call a Python function. (I won’t dwell on how to call the Python parser with a particular string as input — if you’re interested, have a look at the implementation of the \texttt{-c} command line option in \texttt{Modules/main.c} from the Python source code.)

Calling a Python function is easy. First, the Python program must somehow pass you the Python function object. You should provide a function (or some other interface) to do this. When this function is called, save a pointer to the Python function object (be careful to \texttt{Py_INCREF()} it!) in a global variable — or wherever you see fit. For example, the following function might be part of a module definition:

```c
static PyObject *my_callback = NULL;

static PyObject *
my_set_callback(PyObject *dummy, PyObject *args)
{
    PyObject *result = NULL;
    PyObject *temp;

    if (PyArg_ParseTuple(args, "O:set_callback", &temp)) {
        if (!PyCallable_Check(temp)) {
            PyErr_SetString(PyExc_TypeError, "parameter must be
```
This function must be registered with the interpreter using the `METH_VARARGS` flag; this is described in section The Module's Method Table and Initialization Function. The `PyArg_ParseTuple()` function and its arguments are documented in section Extracting Parameters in Extension Functions.

The macros `Py_XINCREF()` and `Py_XDECREF()` increment/decrement the reference count of an object and are safe in the presence of `NULL` pointers (but note that `temp` will not be `NULL` in this context). More info on them in section Reference Counts.

Later, when it is time to call the function, you call the C function `PyObject_CallObject()`. This function has two arguments, both pointers to arbitrary Python objects: the Python function, and the argument list. The argument list must always be a tuple object, whose length is the number of arguments. To call the Python function with no arguments, pass in `NULL`, or an empty tuple; to call it with one argument, pass a singleton tuple. `Py_BuildValue()` returns a tuple when its format string consists of zero or more format codes between parentheses. For example:

```c
int arg;
PyObject *arglist;
PyObject *result;
...
arg = 123;
```
... /* Time to call the callback */
arglist = Py_BuildValue("(i)", arg);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);

PyObject_CallObject() returns a Python object pointer: this is the return value of the Python function. PyObject_CallObject() is “reference-count-neutral” with respect to its arguments. In the example a new tuple was created to serve as the argument list, which is Py_DECREF()-ed immediately after the PyObject_CallObject() call.

The return value of PyObject_CallObject() is “new”: either it is a brand new object, or it is an existing object whose reference count has been incremented. So, unless you want to save it in a global variable, you should somehow Py_DECREF() the result, even (especially!) if you are not interested in its value.

Before you do this, however, it is important to check that the return value isn’t NULL. If it is, the Python function terminated by raising an exception. If the C code that called PyObject_CallObject() is called from Python, it should now return an error indication to its Python caller, so the interpreter can print a stack trace, or the calling Python code can handle the exception. If this is not possible or desirable, the exception should be cleared by calling PyErr_Clear(). For example:

```c
if (result == NULL)
    return NULL; /* Pass error back */
...use result...
Py_DECREF(result);
```

Depending on the desired interface to the Python callback function, you may also have to provide an argument list to PyObject_CallObject(). In some cases the argument list is also
provided by the Python program, through the same interface that specified the callback function. It can then be saved and used in the same manner as the function object. In other cases, you may have to construct a new tuple to pass as the argument list. The simplest way to do this is to call `Py_BuildValue()`. For example, if you want to pass an integral event code, you might use the following code:

```python
PyObject *arglist;
...
arglist = Py_BuildValue("(l)", eventcode);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```

Note the placement of `Py_DECREF(arglist)` immediately after the call, before the error check! Also note that strictly speaking this code is not complete: `Py_BuildValue()` may run out of memory, and this should be checked.

You may also call a function with keyword arguments by using `PyObject_Call()`, which supports arguments and keyword arguments. As in the above example, we use `Py_BuildValue()` to construct the dictionary.

```python
PyObject *dict;
...
dict = Py_BuildValue("{s:i}", "name", val);
result = PyObject_CallObject(my_callback, NULL, dict);
Py_DECREF(dict);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```
1.7. Extracting Parameters in Extension Functions

The `PyArg_ParseTuple()` function is declared as follows:

```c
int PyArg_ParseTuple(PyObject *arg, char *format, ...);
```

The `arg` argument must be a tuple object containing an argument list passed from Python to a C function. The `format` argument must be a format string, whose syntax is explained in Parsing arguments and building values in the Python/C API Reference Manual. The remaining arguments must be addresses of variables whose type is determined by the format string.

Note that while `PyArg_ParseTuple()` checks that the Python arguments have the required types, it cannot check the validity of the addresses of C variables passed to the call: if you make mistakes there, your code will probably crash or at least overwrite random bits in memory. So be careful!

Note that any Python object references which are provided to the caller are borrowed references; do not decrement their reference count!

Some example calls:

```c
int ok;
int i, j;
long k, l;
const char *s;
int size;

ok = PyArg_ParseTuple(args, """); /* No arguments */
   /* Python call: f() */
```
ok = PyArg_ParseTuple(args, "s", &s); /* A string */
   /* Possible Python call: f('whoops!') */
ok = PyArg_ParseTuple(args, "lls", &k, &l, &s); /* Two longs and a string */
   /* Possible Python call: f(1, 2, 'three') */
ok = PyArg_ParseTuple(args, "(ii)s#", &i, &j, &s, &size); /* A pair of ints and a string, whose size is also returned */
   /* Possible Python call: f((1, 2), 'three') */
{
    const char *file;
    const char *mode = "r";
    int bufsize = 0;
    ok = PyArg_ParseTuple(args, "s|si", &file, &mode, &bufsize); /* A string, and optionally another string and an integer */
    /* Possible Python calls:
       f('spam')
       f('spam', 'w')
       f('spam', 'wb', 100000) */
}
{
    int left, top, right, bottom, h, v;
    ok = PyArg_ParseTuple(args, "((ii)(ii))(ii)",
                           &left, &top, &right, &bottom, &h, &v);
    /* A rectangle and a point */
    /* Possible Python call:
       f(((0, 0), (400, 300)), (10, 10)) */
}
{
    Py_complex c;
    ok = PyArg_ParseTuple(args, "D:myfunction", &c);
    /* a complex, also providing a function name for errors */
    /* Possible Python call: myfunction(1+2j) */
}
1.8. Keyword Parameters for Extension Functions

The `PyArg_ParseTupleAndKeywords()` function is declared as follows:

```c
int PyArg_ParseTupleAndKeywords(PyObject *arg, PyObject *kwdict, char *format, char *kwlist[], ...);
```

The `arg` and `format` parameters are identical to those of the `PyArg_ParseTuple()` function. The `kwdict` parameter is the dictionary of keywords received as the third parameter from the Python runtime. The `kwlist` parameter is a `NULL`-terminated list of strings which identify the parameters; the names are matched with the type information from `format` from left to right. On success, `PyArg_ParseTupleAndKeywords()` returns true, otherwise it returns false and raises an appropriate exception.

**Note:** Nested tuples cannot be parsed when using keyword arguments! Keyword parameters passed in which are not present in the `kwlist` will cause `TypeError` to be raised.

Here is an example module which uses keywords, based on an example by Geoff Philbrick (philbrick@hks.com):

```c
#include "Python.h"

static PyObject *
keywdarg_parrot(PyObject *self, PyObject *args, PyObject *keywd)
{
    int voltage;
    char *state = "a stiff";
    char *action = "voom";
    char *type = "Norwegian Blue";

    static char *kwlist[] = {
        "voltage", "state", "action", "typ
if (!PyArg_ParseTupleAndKeywords(args, keywds, "i|sss", kwlist, &voltage, &state, &action, &type, &action)
    return NULL;

printf("-- This parrot wouldn't %s if you put %i Volts through it.
       action, voltage);
printf("-- Lovely plumage, the %s -- It's %s!\n", type, state);
Py_INCREF(Py_None);
    return Py_None;
}

static PyMethodDef keywdarg_methods[] = {
    /* The cast of the function is necessary since PyCFunction
       * only take two PyObject* parameters, and keywdarg_parrot( *
       * three.
       */
    {"parrot", (PyCFunction)keywdarg_parrot, METH_VARARGS | METH_KEYWORDS,
     "Print a lovely skit to standard output."},
    {NULL, NULL, 0, NULL} /* sentinel */
};

void initkeywdarg(void)
{
    /* Create the module and add the functions */
    Py_InitModule("keywdarg", keywdarg_methods);
}
1.9. Building Arbitrary Values

This function is the counterpart to \texttt{PyArg_ParseTuple()}. It is declared as follows:

\begin{verbatim}
PyObject *Py_BuildValue(char *format, ...);
\end{verbatim}

It recognizes a set of format units similar to the ones recognized by \texttt{PyArg_ParseTuple()}, but the arguments (which are input to the function, not output) must not be pointers, just values. It returns a new Python object, suitable for returning from a C function called from Python.

One difference with \texttt{PyArg_ParseTuple()}: while the latter requires its first argument to be a tuple (since Python argument lists are always represented as tuples internally), \texttt{Py_BuildValue()} does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns \texttt{None}; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

Examples (to the left the call, to the right the resulting Python value):

\begin{verbatim}
Py_BuildValue(""")                        None
Py_BuildValue("i", 123)                      123
Py_BuildValue("iii", 123, 456, 789)         (123, 456, 789)
Py_BuildValue("s", "hello")                'hello'
Py_BuildValue("ss", "hello", "world")     ('hello', 'world')
Py_BuildValue("s#", "hello", 4)            'hell'
Py_BuildValue("()")                         ()
Py_BuildValue("(i)", 123)                    (123,)
Py_BuildValue("(ii)", 123, 456)             (123, 456)
Py_BuildValue("(i,i)", 123, 456)            (123, 456)
Py_BuildValue("[i,i]", 123, 456)             [123, 456]
Py_BuildValue("{s:i,s:i}"),              
\end{verbatim}
<table>
<thead>
<tr>
<th>String</th>
<th>Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abc&quot;, 123, &quot;def&quot;, 456</td>
<td>{'abc': 123, 'def': 456}</td>
</tr>
<tr>
<td><code>Py_BuildValue(&quot;((ii)(ii))(ii)&quot;, 1, 2, 3, 4, 5, 6)</code></td>
<td><code>(((1, 2), (3, 4)), (5, 6))</code></td>
</tr>
</tbody>
</table>
1.10. Reference Counts

In languages like C or C++, the programmer is responsible for dynamic allocation and deallocation of memory on the heap. In C, this is done using the functions `malloc()` and `free()`. In C++, the operators `new` and `delete` are used with essentially the same meaning and we'll restrict the following discussion to the C case.

Every block of memory allocated with `malloc()` should eventually be returned to the pool of available memory by exactly one call to `free()`. It is important to call `free()` at the right time. If a block's address is forgotten but `free()` is not called for it, the memory it occupies cannot be reused until the program terminates. This is called a memory leak. On the other hand, if a program calls `free()` for a block and then continues to use the block, it creates a conflict with re-use of the block through another `malloc()` call. This is called using freed memory. It has the same bad consequences as referencing uninitialized data — core dumps, wrong results, mysterious crashes.

Common causes of memory leaks are unusual paths through the code. For instance, a function may allocate a block of memory, do some calculation, and then free the block again. Now a change in the requirements for the function may add a test to the calculation that detects an error condition and can return prematurely from the function. It's easy to forget to free the allocated memory block when taking this premature exit, especially when it is added later to the code. Such leaks, once introduced, often go undetected for a long time: the error exit is taken only in a small fraction of all calls, and most modern machines have plenty of virtual memory, so the leak only becomes apparent in a long-running process that uses the leaking function frequently. Therefore, it's important to prevent leaks
from happening by having a coding convention or strategy that minimizes this kind of errors.

Since Python makes heavy use of `malloc()` and `free()`, it needs a strategy to avoid memory leaks as well as the use of freed memory. The chosen method is called *reference counting*. The principle is simple: every object contains a counter, which is incremented when a reference to the object is stored somewhere, and which is decremented when a reference to it is deleted. When the counter reaches zero, the last reference to the object has been deleted and the object is freed.

An alternative strategy is called *automatic garbage collection*. (Sometimes, reference counting is also referred to as a garbage collection strategy, hence my use of “automatic” to distinguish the two.) The big advantage of automatic garbage collection is that the user doesn’t need to call `free()` explicitly. (Another claimed advantage is an improvement in speed or memory usage — this is no hard fact however.) The disadvantage is that for C, there is no truly portable automatic garbage collector, while reference counting can be implemented portably (as long as the functions `malloc()` and `free()` are available — which the C Standard guarantees). Maybe some day a sufficiently portable automatic garbage collector will be available for C. Until then, we’ll have to live with reference counts.

While Python uses the traditional reference counting implementation, it also offers a cycle detector that works to detect reference cycles. This allows applications to not worry about creating direct or indirect circular references; these are the weakness of garbage collection implemented using only reference counting. Reference cycles consist of objects which contain (possibly indirect) references to themselves, so that each object in the cycle has a reference count which is non-zero. Typical reference counting implementations are not able to reclaim the memory belonging to any objects in a
reference cycle, or referenced from the objects in the cycle, even though there are no further references to the cycle itself.

The cycle detector is able to detect garbage cycles and can reclaim them so long as there are no finalizers implemented in Python (__del__() methods). When there are such finalizers, the detector exposes the cycles through the gc module (specifically, the garbage variable in that module). The gc module also exposes a way to run the detector (the collect() function), as well as configuration interfaces and the ability to disable the detector at runtime. The cycle detector is considered an optional component; though it is included by default, it can be disabled at build time using the --without-cycle-gc option to the configure script on Unix platforms (including Mac OS X) or by removing the definition of WITH_CYCLE_GC in the pyconfig.h header on other platforms. If the cycle detector is disabled in this way, the gc module will not be available.

1.10.1. Reference Counting in Python

There are two macros, Py_INCREF(x) and Py_DECREF(x), which handle the incrementing and decrementing of the reference count. Py_DECREF() also frees the object when the count reaches zero. For flexibility, it doesn’t call free() directly — rather, it makes a call through a function pointer in the object’s type object. For this purpose (and others), every object also contains a pointer to its type object.

The big question now remains: when to use Py_INCREF(x) and Py_DECREF(x)? Let’s first introduce some terms. Nobody “owns” an object; however, you can own a reference to an object. An object’s reference count is now defined as the number of owned references to it. The owner of a reference is responsible for calling Py_DECREF() when the reference is no longer needed. Ownership of a reference
can be transferred. There are three ways to dispose of an owned reference: pass it on, store it, or call `Py_DECREF()`. Forgetting to dispose of an owned reference creates a memory leak.

It is also possible to borrow [2] a reference to an object. The borrower of a reference should not call `Py_DECREF()`. The borrower must not hold on to the object longer than the owner from which it was borrowed. Using a borrowed reference after the owner has disposed of it risks using freed memory and should be avoided completely. [3]

The advantage of borrowing over owning a reference is that you don’t need to take care of disposing of the reference on all possible paths through the code — in other words, with a borrowed reference you don’t run the risk of leaking when a premature exit is taken. The disadvantage of borrowing over owning is that there are some subtle situations where in seemingly correct code a borrowed reference can be used after the owner from which it was borrowed has in fact disposed of it.

A borrowed reference can be changed into an owned reference by calling `Py_INCREF()`. This does not affect the status of the owner from which the reference was borrowed — it creates a new owned reference, and gives full owner responsibilities (the new owner must dispose of the reference properly, as well as the previous owner).

**1.10.2. Ownership Rules**

Whenever an object reference is passed into or out of a function, it is part of the function’s interface specification whether ownership is transferred with the reference or not.

Most functions that return a reference to an object pass on ownership with the reference. In particular, all functions whose function it is to create a new object, such as `PyInt_FromLong()` and
Py_BuildValue(), pass ownership to the receiver. Even if the object is not actually new, you still receive ownership of a new reference to that object. For instance, PyInt_FromLong() maintains a cache of popular values and can return a reference to a cached item.

Many functions that extract objects from other objects also transfer ownership with the reference, for instance PyObject_GetAttrString(). The picture is less clear, here, however, since a few common routines are exceptions: PyTuple_GetItem(), PyList_GetItem(), PyDict_GetItem(), and PyDict_GetItemString() all return references that you borrow from the tuple, list or dictionary.

The function PyImport_AddModule() also returns a borrowed reference, even though it may actually create the object it returns: this is possible because an owned reference to the object is stored in sys.modules.

When you pass an object reference into another function, in general, the function borrows the reference from you — if it needs to store it, it will use Py_INCREF() to become an independent owner. There are exactly two important exceptions to this rule: PyTuple_SetItem() and PyList_SetItem(). These functions take over ownership of the item passed to them — even if they fail! (Note that PyDict_SetItem() and friends don’t take over ownership — they are “normal.”)

When a C function is called from Python, it borrows references to its arguments from the caller. The caller owns a reference to the object, so the borrowed reference’s lifetime is guaranteed until the function returns. Only when such a borrowed reference must be stored or passed on, it must be turned into an owned reference by calling Py_INCREF().

The object reference returned from a C function that is called from Python must be an owned reference — ownership is transferred
from the function to its caller.

## 1.10.3. Thin Ice

There are a few situations where seemingly harmless use of a borrowed reference can lead to problems. These all have to do with implicit invocations of the interpreter, which can cause the owner of a reference to dispose of it.

The first and most important case to know about is using `Py_DECREF()` on an unrelated object while borrowing a reference to a list item. For instance:

```c
void bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);
    PyList_SetItem(list, 1, PyInt_FromLong(0L));
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

This function first borrows a reference to `list[0]`, then replaces `list[1]` with the value `0`, and finally prints the borrowed reference. Looks harmless, right? But it's not!

Let’s follow the control flow into `PyList_SetItem()`. The list owns references to all its items, so when item 1 is replaced, it has to dispose of the original item 1. Now let’s suppose the original item 1 was an instance of a user-defined class, and let’s further suppose that the class defined a `__del__()` method. If this class instance has a reference count of 1, disposing of it will call its `__del__()` method.

Since it is written in Python, the `__del__()` method can execute arbitrary Python code. Could it perhaps do something to invalidate the reference to `item` in `bug()`? You bet! Assuming that the list
passed into `bug()` is accessible to the `__del__()` method, it could execute a statement to the effect of `del list[0]`, and assuming this was the last reference to that object, it would free the memory associated with it, thereby invalidating `item`.

The solution, once you know the source of the problem, is easy: temporarily increment the reference count. The correct version of the function reads:

```c
void no_bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);
    Py_INCREF(item);
    PyList_SetItem(list, 1, PyInt_FromLong(0L));
    PyObject_Print(item, stdout, 0);
    Py_DECREF(item);
}
```

This is a true story. An older version of Python contained variants of this bug and someone spent a considerable amount of time in a C debugger to figure out why his `__del__()` methods would fail...

The second case of problems with a borrowed reference is a variant involving threads. Normally, multiple threads in the Python interpreter can’t get in each other’s way, because there is a global lock protecting Python’s entire object space. However, it is possible to temporarily release this lock using the macro `Py_BEGIN_ALLOW_THREADS`, and to re-acquire it using `Py_END_ALLOW_THREADS`. This is common around blocking I/O calls, to let other threads use the processor while waiting for the I/O to complete. Obviously, the following function has the same problem as the previous one:

```c
void bug(PyObject *list)
```
In general, functions that take object references as arguments do not expect you to pass them NULL pointers, and will dump core (or cause later core dumps) if you do so. Functions that return object references generally return NULL only to indicate that an exception occurred. The reason for not testing for NULL arguments is that functions often pass the objects they receive on to other function — if each function were to test for NULL, there would be a lot of redundant tests and the code would run more slowly.

It is better to test for NULL only at the “source:” when a pointer that may be NULL is received, for example, from malloc() or from a function that may raise an exception.

The macros Py_INCREF() and Py_DECREF() do not check for NULL pointers — however, their variants Py_XINCREF() and Py_XDECREF() do.

The macros for checking for a particular object type (Pytype_Check()) don’t check for NULL pointers — again, there is much code that calls several of these in a row to test an object against various different expected types, and this would generate redundant tests. There are no variants with NULL checking.

The C function calling mechanism guarantees that the argument list passed to C functions (args in the examples) is never NULL — in fact it guarantees that it is always a tuple.
It is a severe error to ever let a NULL pointer “escape” to the Python user.
1.11. Writing Extensions in C++

It is possible to write extension modules in C++. Some restrictions apply. If the main program (the Python interpreter) is compiled and linked by the C compiler, global or static objects with constructors cannot be used. This is not a problem if the main program is linked by the C++ compiler. Functions that will be called by the Python interpreter (in particular, module initialization functions) have to be declared using `extern "C"`. It is unnecessary to enclose the Python header files in `extern "C" {...}` — they use this form already if the symbol `__cplusplus` is defined (all recent C++ compilers define this symbol).
1.12. Providing a C API for an Extension Module

Many extension modules just provide new functions and types to be used from Python, but sometimes the code in an extension module can be useful for other extension modules. For example, an extension module could implement a type “collection” which works like lists without order. Just like the standard Python list type has a C API which permits extension modules to create and manipulate lists, this new collection type should have a set of C functions for direct manipulation from other extension modules.

At first sight this seems easy: just write the functions (without declaring them static, of course), provide an appropriate header file, and document the C API. And in fact this would work if all extension modules were always linked statically with the Python interpreter. When modules are used as shared libraries, however, the symbols defined in one module may not be visible to another module. The details of visibility depend on the operating system; some systems use one global namespace for the Python interpreter and all extension modules (Windows, for example), whereas others require an explicit list of imported symbols at module link time (AIX is one example), or offer a choice of different strategies (most Unices). And even if symbols are globally visible, the module whose functions one wishes to call might not have been loaded yet!

Portability therefore requires not to make any assumptions about symbol visibility. This means that all symbols in extension modules should be declared static, except for the module’s initialization function, in order to avoid name clashes with other extension modules (as discussed in section The Module’s Method Table and Initialization Function). And it means that symbols that should be accessible from other extension modules must be exported in a
different way.

Python provides a special mechanism to pass C-level information (pointers) from one extension module to another one: Capsules. A Capsule is a Python data type which stores a pointer (void *). Capsules can only be created and accessed via their C API, but they can be passed around like any other Python object. In particular, they can be assigned to a name in an extension module’s namespace. Other extension modules can then import this module, retrieve the value of this name, and then retrieve the pointer from the Capsule.

There are many ways in which Capsules can be used to export the C API of an extension module. Each function could get its own Capsule, or all C API pointers could be stored in an array whose address is published in a Capsule. And the various tasks of storing and retrieving the pointers can be distributed in different ways between the module providing the code and the client modules.

Whichever method you choose, it's important to name your Capsules properly. The function `PyCapsule_New()` takes a name parameter (const char *); you're permitted to pass in a NULL name, but we strongly encourage you to specify a name. Properly named Capsules provide a degree of runtime type-safety; there is no feasible way to tell one unnamed Capsule from another.

In particular, Capsules used to expose C APIs should be given a name following this convention:

```
modulename.attributename
```

The convenience function `PyCapsule_Import()` makes it easy to load a C API provided via a Capsule, but only if the Capsule’s name matches this convention. This behavior gives C API users a high degree of certainty that the Capsule they load contains the correct C
The following example demonstrates an approach that puts most of the burden on the writer of the exporting module, which is appropriate for commonly used library modules. It stores all C API pointers (just one in the example!) in an array of `void` pointers which becomes the value of a Capsule. The header file corresponding to the module provides a macro that takes care of importing the module and retrieving its C API pointers; client modules only have to call this macro before accessing the C API.

The exporting module is a modification of the `spam` module from section A Simple Example. The function `spam.system()` does not call the C library function `system()` directly, but a function `PySpam_System()`, which would of course do something more complicated in reality (such as adding “spam” to every command). This function `PySpam_System()` is also exported to other extension modules.

The function `PySpam_System()` is a plain C function, declared `static` like everything else:

```c
static int
PySpam_System(const char *command)
{
    return system(command);
}
```

The function `spam_system()` is modified in a trivial way:

```c
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
```
In the beginning of the module, right after the line

```
#include "Python.h"
```

two more lines must be added:

```
#define SPAM_MODULE
#include "spammodule.h"
```

The `#define` is used to tell the header file that it is being included in the exporting module, not a client module. Finally, the module's initialization function must take care of initializing the C API pointer array:

```c
PyMODINIT_FUNC
initspam(void)
{
    PyObject *m;
    static void *PySpam_API[PySpam_API_pointers];
    PyObject *c_api_object;

    m = Py_InitModule("spam", SpamMethods);
    if (m == NULL)
        return;

    /* Initialize the C API pointer array */
    PySpam_API[PySpam_System_NUM] = (void *)PySpam_System;

    /* Create a Capsule containing the API pointer array's addr */
    c_api_object = PyCapsule_New((void *)PySpam_API, "spam._C_API",
                                 (void *)PySpam_API_pointers);
    if (c_api_object != NULL)
        PyModule_AddObject(m, "_C_API", c_api_object);
}
```

Note that `PySpam_API` is declared `static`; otherwise the pointer array
would disappear when \texttt{initspam()} terminates!

The bulk of the work is in the header file \texttt{spammodule.h}, which looks like this:

```c
#ifndef Py_SPAMMODULE_H
#define Py_SPAMMODULE_H
#ifdef __cplusplus
extern "C" {
#endif

/* Header file for spammodule */

/* C API functions */
#define PySpam_System_NUM 0
#define PySpam_System_RETURN int
#define PySpam_System_PROTO (const char *command)

/* Total number of C API pointers */
#define PySpam_API_pointers 1

#ifdef SPAM_MODULE
/* This section is used when compiling spammodule.c */

static PySpam_System_RETURN PySpam_System PySpam_System_PROTO;
#else
/* This section is used in modules that use spammodule's API */

static void **PySpam_API;

#define PySpam_System \
  (*(PySpam_System_RETURN (*)PySpam_System_PROTO) PySpam_API[PyS

/* Return -1 on error, 0 on success. 
 * PyCapsule_Import will set an exception if there's an error. */

static int import_spam(void)
{
  PySpam_API = (void **)PyCapsule_Import("spam._C_API", 0);
  return (PySpam_API != NULL) ? 0 : -1;
}
```
```
All that a client module must do in order to have access to the function `PySpam_System()` is to call the function (or rather macro) `import_spam()` in its initialization function:

```python
PyMODINIT_FUNC
initclient(void)
{
    PyObject *m;

    m = Py_InitModule("client", ClientMethods);
    if (m == NULL)
        return;
    if (import_spam() < 0)
        return;
    /* additional initialization can happen here */
}
```

The main disadvantage of this approach is that the file `spammodule.h` is rather complicated. However, the basic structure is the same for each function that is exported, so it has to be learned only once.

Finally it should be mentioned that Capsules offer additional functionality, which is especially useful for memory allocation and deallocation of the pointer stored in a Capsule. The details are described in the Python/C API Reference Manual in the section Capsules and in the implementation of Capsules (files `Include/pycapsule.h` and `Objects/pycapsule.c` in the Python source code distribution).

Footnotes
An interface for this function already exists in the standard module `os` — it was chosen as a simple and straightforward example.

The metaphor of “borrowing” a reference is not completely correct: the owner still has a copy of the reference. Checking that the reference count is at least 1 **does not work** — the reference count itself could be in freed memory and may thus be reused for another object!

These guarantees don’t hold when you use the “old” style calling convention — this is still found in much existing code.
2. Defining New Types

As mentioned in the last chapter, Python allows the writer of an extension module to define new types that can be manipulated from Python code, much like strings and lists in core Python.

This is not hard; the code for all extension types follows a pattern, but there are some details that you need to understand before you can get started.

**Note:** The way new types are defined changed dramatically (and for the better) in Python 2.2. This document documents how to define new types for Python 2.2 and later. If you need to support older versions of Python, you will need to refer to older versions of this documentation.
2.1. The Basics

The Python runtime sees all Python objects as variables of type `PyObject*`. A `PyObject` is not a very magnificent object - it just contains the refcount and a pointer to the object’s “type object”. This is where the action is; the type object determines which (C) functions get called when, for instance, an attribute gets looked up on an object or it is multiplied by another object. These C functions are called “type methods”.

So, if you want to define a new object type, you need to create a new type object.

This sort of thing can only be explained by example, so here’s a minimal, but complete, module that defines a new type:

```c
#include <Python.h>

typedef struct {
    PyObject_HEAD
    /* Type-specific fields go here. */
} noddy_NoddyObject;

static PyTypeObject noddy_NoddyType = {
    PyObject_HEAD_INIT(NULL)
    0,           /*ob_size*/
    "noddy.Noddy",
    sizeof(noddy_NoddyObject), /*tp_basicsize*/
    0,           /*tp_itemsize*/
    0,           /*tp_dealloc*/
    0,           /*tp_print*/
    0,           /*tp_getattr*/
    0,           /*tp_setattr*/
    0,           /*tp_compare*/
    0,           /*tp_as_number*/
    0,           /*tp_as_sequence*/
    0,           /*tp_as_mapping*/
    0,           /*tp_hash */
    0,           /*tp_call*/
```
Now that's quite a bit to take in at once, but hopefully bits will seem familiar from the last chapter.

The first bit that will be new is:

```c
typedef struct {
    PyObject_HEAD
} noddy_NoddyObject;
```

This is what a Noddy object will contain—in this case, nothing more than every Python object contains, namely a refcount and a pointer to a type object. These are the fields the `PyObject_HEAD` macro brings
in. The reason for the macro is to standardize the layout and to enable special debugging fields in debug builds. Note that there is no semicolon after the `PyObject_HEAD` macro; one is included in the macro definition. Be wary of adding one by accident; it’s easy to do from habit, and your compiler might not complain, but someone else’s probably will! (On Windows, MSVC is known to call this an error and refuse to compile the code.)

For contrast, let’s take a look at the corresponding definition for standard Python integers:

```c
typedef struct {
    PyObject_HEAD
    long ob_ival;
} PyIntObject;
```

Moving on, we come to the crunch — the type object.

```c
static PyTypeObject noddy_NoddyType = {
    PyObject_HEAD_INIT(NULL)
    0, /*ob_size*/
    "noddy.Noddy", /*tp_name*/
    sizeof(noddy_NoddyObject), /*tp_basicsize*/
    0, /*tp_itemsize*/
    0, /*tp_dealloc*/
    0, /*tp_print*/
    0, /*tp_getattr*/
    0, /*tp_setattr*/
    0, /*tp_compare*/
    0, /*tp_repr*/
    0, /*tp_as_number*/
    0, /*tp_as_sequence*/
    0, /*tp_as_mapping*/
    0, /*tp_hash */
    0, /*tp_call*/
    0, /*tp_str*/
    0, /*tp_getattro*/
    0, /*tp_setattro*/
    0, /*tp_as_buffer*/
    Py_TPFLAGS_DEFAULT, /*tp_flags*/
    "Noddy objects", /* tp_doc */
};
```
Now if you go and look up the definition of `PyTypeObject` in `object.h` you'll see that it has many more fields that the definition above. The remaining fields will be filled with zeros by the C compiler, and it's common practice to not specify them explicitly unless you need them.

This is so important that we're going to pick the top of it apart still further:

```c
PyObject_HEAD_INIT(NULL)
```

This line is a bit of a wart; what we'd like to write is:

```c
PyObject_HEAD_INIT(&PyType_Type)
```

as the type of a type object is “type”, but this isn't strictly conforming C and some compilers complain. Fortunately, this member will be filled in for us by `PyType_Ready()`.

```c
0, /* ob_size */
```

The `ob_size` field of the header is not used; its presence in the type structure is a historical artifact that is maintained for binary compatibility with extension modules compiled for older versions of Python. Always set this field to zero.

```c
"noddy.Noddy", /* tp_name */
```

The name of our type. This will appear in the default textual representation of our objects and in some error messages, for example:

```python
>>> "" + noddy.new_noddy()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot add type "noddy.Noddy" to string
```
Note that the name is a dotted name that includes both the module name and the name of the type within the module. The module in this case is `noddy` and the type is `Noddy`, so we set the type name to `noddy.Noddy`. One side effect of using an undotted name is that the pydoc documentation tool will not list the new type in the module documentation.

```c
sizeof(noddy_NoddyObject), /* tp_baseasize */
```

This is so that Python knows how much memory to allocate when you call `PyObject_New()`.

**Note:** If you want your type to be subclassable from Python, and your type has the same `tp_baseasize` as its base type, you may have problems with multiple inheritance. A Python subclass of your type will have to list your type first in its `__bases__`, or else it will not be able to call your type’s `__new__()` method without getting an error. You can avoid this problem by ensuring that your type has a larger value for `tp_baseasize` than its base type does. Most of the time, this will be true anyway, because either your base type will be `object`, or else you will be adding data members to your base type, and therefore increasing its size.

```c
0, /* tp_itemsize */
```

This has to do with variable length objects like lists and strings. Ignore this for now.

Skipping a number of type methods that we don’t provide, we set the class flags to `Py_TPFLAGS_DEFAULT`.

```c
Py_TPFLAGS_DEFAULT, /*tp_flags*/
```

All types should include this constant in their flags. It enables all of
the members defined by the current version of Python.

We provide a doc string for the type in `tp_doc`.

```
"Noddy objects", /* tp_doc */
```

Now we get into the type methods, the things that make your objects different from the others. We aren't going to implement any of these in this version of the module. We'll expand this example later to have more interesting behavior.

For now, all we want to be able to do is to create new `Noddy` objects. To enable object creation, we have to provide a `tp_new` implementation. In this case, we can just use the default implementation provided by the API function `PyType_GenericNew()`. We'd like to just assign this to the `tp_new` slot, but we can't, for portability sake, On some platforms or compilers, we can't statically initialize a structure member with a function defined in another C module, so, instead, we'll assign the `tp_new` slot in the module initialization function just before calling `PyType_Ready()`:

```c
noddy_NoddyType.tp_new = PyType_GenericNew;
if (PyType_Ready(&noddy_NoddyType) < 0)
    return;
```

All the other type methods are `NULL`, so we'll go over them later — that's for a later section!

Everything else in the file should be familiar, except for some code in `initnoddy()`:

```c
if (PyType_Ready(&noddy_NoddyType) < 0)
    return;
```

This initializes the `Noddy` type, filing in a number of members,
including `ob_type` that we initially set to `NULL`.

```c
PyModule_AddObject(m, "Noddy", (PyObject *)&noddy_NoddyType);
```

This adds the type to the module dictionary. This allows us to create `Noddy` instances by calling the `Noddy` class:

```python
>>> import noddy
>>> mynoddy = noddy.Noddy()
```

That's it! All that remains is to build it; put the above code in a file called `noddy.c` and

```python
from distutils.core import setup, Extension
setup(name="noddy", version="1.0",
      ext_modules=[Extension("noddy", ["noddy.c"])]
)
```

in a file called `setup.py`; then typing

```
$ python setup.py build
```

at a shell should produce a file `noddy.so` in a subdirectory; move to that directory and fire up Python — you should be able to import `noddy` and play around with Noddy objects.

That wasn't so hard, was it?

Of course, the current Noddy type is pretty uninteresting. It has no data and doesn't do anything. It can't even be subclassed.

### 2.1.1. Adding data and methods to the Basic example

Let's expend the basic example to add some data and methods. Let's also make the type usable as a base class. We'll create a new
module, noddy2 that adds these capabilities:

```c
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    PyObject *first;  /* first name */
    PyObject *last;  /* last name */
    int number;
} Noddy;

static void
Noddy_dealloc(Noddy* self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    self->ob_type->tp_free((PyObject*)self);
}

static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
    }

    self->last = PyString_FromString("");
    if (self->last == NULL) {
        Py_DECREF(self);
        return NULL;
    }

    self->number = 0;

    return (PyObject *)self;
}
```c
static int Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", kwlist
                                        &first, &last,
                                        &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }

    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }

    return 0;
}

static PyMemberDef Noddy_members[] = {
    {"first", T_OBJECT_EX, offsetof(Noddy, first), 0,
        "first name"},
    {"last", T_OBJECT_EX, offsetof(Noddy, last), 0,
        "last name"},
    {"number", T_INT, offsetof(Noddy, number), 0,
        "noddy number"},
    {NULL} /* Sentinel */
};

static PyObject * Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;
```
if (format == NULL) {
    format = PyString_FromString("%s %s");
    if (format == NULL)
        return NULL;
}

if (self->first == NULL) {
    PyErr_SetString(PyExc_AttributeError, "first");
    return NULL;
}

if (self->last == NULL) {
    PyErr_SetString(PyExc_AttributeError, "last");
    return NULL;
}

args = Py_BuildValue("OO", self->first, self->last);
if (args == NULL)
    return NULL;

result = PyString_Format(format, args);
Py_DECREF(args);

return result;

static PyMethodDef Noddy_methods[] = {
    {"name", (PyCFunction)Noddy_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};

static PyTypeObject NoddyType = {
    PyObject_HEAD_INIT(NULL)
    0,                                    /*ob_size*/
    "noddy.Noddy",                        /*tp_name*/
    sizeof(Noddy),                       /*tp_basicsize*/
    0,                                    /*tp_itemsize*/
    (destructor)Noddy_dealloc,            /*tp_dealloc*/
    0,                                    /*tp_print*/
    0,                                    /*tp_getattr*/
    0,                                    /*tp_setattr*/
    0,                                    /*tp_compare*/
    0,                                    /*tp_repr*/
    0,                                    /*tp_as_number*/
    0,                                    /*tp_as_sequence*/
    0,                                    /*tp_as_mapping*/
    0,                                    /*tp_hashval*/
    0,                                    /*tp_accept*/
    0,                                    /*tp_is_gc*/
    0,                                    /*tp_bases*/
    0,                                    /*tp_mro*/
    0,                                    /*tp_cache*/
    0,                                    /*tp�名*
static PyMethodDef module_methods[] = {
    {NULL} /* Sentinel */
};

#ifndef PyMODINIT_FUNC /* declarations for DLL import/export */
#define PyMODINIT_FUNC void
#endif
PyMODINIT_FUNC
initnoddy2(void)
{
    PyObject* m;

    if (PyType_Ready(&NoddyType) < 0)
        return;

    m = Py_InitModule3("noddy2", module_methods,
                       "Example module that creates an extension type.");

    if (m == NULL)
This version of the module has a number of changes.

We’ve added an extra include:

```c
#include <structmember.h>
```

This include provides declarations that we use to handle attributes, as described a bit later.

The name of the `Noddy` object structure has been shortened to `Noddy`. The type object name has been shortened to `NoddyType`.

The `Noddy` type now has three data attributes, `first`, `last`, and `number`. The `first` and `last` variables are Python strings containing first and last names. The `number` attribute is an integer.

The object structure is updated accordingly:

```c
typedef struct {
    PyObject_HEAD
    PyObject *first;
    PyObject *last;
    int number;
} Noddy;
```

Because we now have data to manage, we have to be more careful about object allocation and deallocation. At a minimum, we need a deallocation method:

```c
static void
Noddy_dealloc(Noddy* self)
{ return;
    Py_INCREF(&NoddyType);
    PyModule_AddObject(m, "Noddy", (PyObject *)&NoddyType);
}
which is assigned to the \texttt{tp\_dealloc} member:

\begin{verbatim}
(destructor)Noddy\_dealloc, /*tp\_dealloc*/
\end{verbatim}

This method decrements the reference counts of the two Python attributes. We use \texttt{Py\_XDECREF()} here because the \texttt{first} and \texttt{last} members could be \texttt{NULL}. It then calls the \texttt{tp\_free} member of the object's type to free the object's memory. Note that the object's type might not be \texttt{NoddyType}, because the object may be an instance of a subclass.

We want to make sure that the first and last names are initialized to empty strings, so we provide a new method:

\begin{verbatim}
static PyObject *
Noddy\_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;
    self = (Noddy *)type->tp\_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString\_From\_String(""");
        if (self->first == NULL)
            \{ Py\_DECREF(self);
            return NULL;
        \}
        self->last = PyString\_From\_String(""");
        if (self->last == NULL)
            \{ Py\_DECREF(self);
            return NULL;
        \}
        self->number = 0;
    }
}\end{verbatim}
```c
return (PyObject *)self;
```

and install it in the `tp_new` member:

```c
Noddy_new,   /* tp_new */
```

The new member is responsible for creating (as opposed to initializing) objects of the type. It is exposed in Python as the `__new__()` method. See the paper titled “Unifying types and classes in Python” for a detailed discussion of the `__new__()` method. One reason to implement a new method is to assure the initial values of instance variables. In this case, we use the new method to make sure that the initial values of the members `first` and `last` are not `NULL`. If we didn’t care whether the initial values were `NULL`, we could have used `PyType_GenericNew()` as our new method, as we did before. `PyType_GenericNew()` initializes all of the instance variable members to `NULL`.

The new method is a static method that is passed the type being instantiated and any arguments passed when the type was called, and that returns the new object created. New methods always accept positional and keyword arguments, but they often ignore the arguments, leaving the argument handling to initializer methods. Note that if the type supports subclassing, the type passed may not be the type being defined. The new method calls the `tp_alloc` slot to allocate memory. We don’t fill the `tp_alloc` slot ourselves. Rather `PyType_Ready()` fills it for us by inheriting it from our base class, which is `object` by default. Most types use the default allocation.

**Note:** If you are creating a co-operative `tp_new` (one that calls a base type’s `tp_new` or `__new__()`, you must *not* try to determine
what method to call using method resolution order at runtime. Always statically determine what type you are going to call, and call its `tp_new` directly, or via `type->tp_base->tp_new`. If you do not do this, Python subclasses of your type that also inherit from other Python-defined classes may not work correctly. (Specifically, you may not be able to create instances of such subclasses without getting a `TypeError`.)

We provide an initialization function:

```c
static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", kwlist
                                        &first, &last,
                                        &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCRREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }

    if (last) {
        tmp = self->last;
        Py_INCRREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }

    return 0;
}
```

by filling the `tp_init` slot.
The `tp_init` slot is exposed in Python as the `__init__()` method. It is used to initialize an object after it's created. Unlike the new method, we can't guarantee that the initializer is called. The initializer isn't called when unpickling objects and it can be overridden. Our initializer accepts arguments to provide initial values for our instance. Initializers always accept positional and keyword arguments.

Initializers can be called multiple times. Anyone can call the `__init__()` method on our objects. For this reason, we have to be extra careful when assigning the new values. We might be tempted, for example to assign the `first` member like this:

```python
if (first) {
    Py_XDECREF(self->first);
    Py_INCREF(first);
    self->first = first;
}
```

But this would be risky. Our type doesn't restrict the type of the `first` member, so it could be any kind of object. It could have a destructor that causes code to be executed that tries to access the `first` member. To be paranoid and protect ourselves against this possibility, we almost always reassign members before decrementing their reference counts. When don't we have to do this?

- when we absolutely know that the reference count is greater than 1
- when we know that deallocation of the object [1] will not cause any calls back into our type's code
- when decrementing a reference count in a `tp_dealloc` handler when garbage-collections is not supported [2]

We want to expose our instance variables as attributes. There are a number of ways to do that. The simplest way is to define member
definitions:

```c
static PyMemberDef Noddy_members[] = {
    {"first", T_OBJECT_EX, offsetof(Noddy, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(Noddy, last), 0,
     "last name"},
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL} /* Sentinel */
};
```

and put the definitions in the `tp_members` slot:

```
Noddy_members, /* tp_members */
```

Each member definition has a member name, type, offset, access flags and documentation string. See the [Generic Attribute Management](#) section below for details.

A disadvantage of this approach is that it doesn’t provide a way to restrict the types of objects that can be assigned to the Python attributes. We expect the first and last names to be strings, but any Python objects can be assigned. Further, the attributes can be deleted, setting the C pointers to `NULL`. Even though we can make sure the members are initialized to non-`NULL` values, the members can be set to `NULL` if the attributes are deleted.

We define a single method, `name()`, that outputs the objects name as the concatenation of the first and last names.

```c
static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
        format = PyString_FromString("%s %s");

```
The method is implemented as a C function that takes a Noddy (or Noddy subclass) instance as the first argument. Methods always take an instance as the first argument. Methods often take positional and keyword arguments as well, but in this case we don’t take any and don’t need to accept a positional argument tuple or keyword argument dictionary. This method is equivalent to the Python method:

```python
def name(self):
    return "%s %s" % (self.first, self.last)
```

Note that we have to check for the possibility that our `first` and `last` members are `NULL`. This is because they can be deleted, in which case they are set to `NULL`. It would be better to prevent deletion of these attributes and to restrict the attribute values to be strings. We'll see how to do that in the next section.
Now that we've defined the method, we need to create an array of method definitions:

```c
static PyMethodDef Noddy_methods[] = {
    {"name", (PyCFunction)Noddy_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};
```

and assign them to the `tp_methods` slot:

```c
Noddy_methods,  /* tp_methods */
```

Note that we used the `METH_NOARGS` flag to indicate that the method is passed no arguments.

Finally, we'll make our type usable as a base class. We've written our methods carefully so far so that they don't make any assumptions about the type of the object being created or used, so all we need to do is to add the `Py_TPFLAGS_BASETYPE` to our class flag definition:

```c
Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /*tp_flags*/
```

We rename `initnoddy()` to `initnoddy2()` and update the module name passed to `Py_InitModule3()`.

Finally, we update our `setup.py` file to build the new module:

```python
from distutils.core import setup, Extension
setup(name="noddy", version="1.0",
    ext_modules=[
        Extension("noddy", ["noddy.c"]),
        Extension("noddy2", ["noddy2.c"])])
```
In this section, we’ll provide finer control over how the `first` and `last` attributes are set in the `Noddy` example. In the previous version of our module, the instance variables `first` and `last` could be set to non-string values or even deleted. We want to make sure that these attributes always contain strings.

```c
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    PyObject *first;
    PyObject *last;
    int number;
} Noddy;

static void
Noddy_dealloc(Noddy * self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    self->ob_type->tp_free((PyObject*)self);
}

static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("\n");
        if (self->first == NULL)
        {
            Py_DECREF(self);
            return NULL;
        }

        self->last = PyString_FromString("\n");
        if (self->last == NULL)
        {
```
Py_DECREF(self);
    return NULL;
}

self->number = 0;
}

return (PyObject *)self;

static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;
    static char *kwlist[] = {"first", "last", "number", NULL};

    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|SSi", kwlis
     &first, &last, &self->number))
    {
        return -1;
    }

        if (first) {
            tmp = self->first;
            Py_INCREF(first);
            self->first = first;
            Py_DECREF(tmp);
        }

        if (last) {
            tmp = self->last;
            Py_INCREF(last);
            self->last = last;
            Py_DECREF(tmp);
        }

        return 0;
}

static PyMemberDef Noddy_members[] = {
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL} /* Sentinel */
};

static PyObject *
Noddy_getfirst(Noddy *self, void *closure)
```c
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Noddy_setfirst(Noddy *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }

    if (! PyString_Check(value)) {
        PyErr_SetString(PyExc_TypeError, "The first attribute value must be a string");
        return -1;
    }

    Py_DECREF(self->first);
    Py_INCREF(value);
    self->first = value;

    return 0;
}

static PyObject *
Noddy_getlast(Noddy *self, void *closure)
{
    Py_INCREF(self->last);
    return self->last;
}

static int
Noddy_setlast(Noddy *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }

    if (! PyString_Check(value)) {
        PyErr_SetString(PyExc_TypeError, "The last attribute value must be a string");
        return -1;
    }
```
Py_DECREF(self->last);
Py_INCREF(value);
self->last = value;

return 0;
}

static PyGetSetDef Noddy_getseters[] = {
    {"first",
     (getter)Noddy_getfirst, (setter)Noddy_setfirst,
     "first name",
     NULL},
    {"last",
     (getter)Noddy_getlast, (setter)Noddy_setlast,
     "last name",
     NULL},
    {NULL} /* Sentinel */
};

static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
        format = PyString_FromString("%s %s");
        if (format == NULL)
            return NULL;
    }

    args = Py_BuildValue("OO", self->first, self->last);
    if (args == NULL)
        return NULL;

    result = PyString_Format(format, args);
    Py_DECREF(args);

    return result;
}

static PyMethodDef Noddy_methods[] = {
    {"name", (PyCFunction)Noddy_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};
static PyTypeObject NoddyType = {
    PyObject_HEAD_INIT(NULL)
    0, /*ob_size*/
    "noddy.Noddy", /*tp_name*/
    sizeof(Noddy), /*tp_basicsize*/
    0, /*tp_itemsize*/
    (destroyer)Noddy_dealloc, /*tp_dealloc*/
    0, /*tp_print*/
    0, /*tp_getattr*/
    0, /*tp_setattr*/
    0, /*tp_compare*/
    0, /*tp_repr*/
    0, /*tp_as_number*/
    0, /*tp_as_sequence*/
    0, /*tp_as_mapping*/
    0, /*tp_hash */
    0, /*tp_call*/
    0, /*tp_str*/
    0, /*tp_getattro*/
    0, /*tp_setattro*/
    0, /*tp_as_buffer*/
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /*tp_flags*/
    "Noddy objects", /* tp_doc */
    0, /* tp_traverse */
    0, /* tp_clear */
    0, /* tp_richcompare */
    0, /* tp_weaklistoffset */
    0, /* tp_iter */
    0, /* tp_iternext */
    Noddy_methods, /* tp_methods */
    Noddy_members, /* tp_members */
    Noddy_getseters, /* tp_getset */
    0, /* tp_base */
    0, /* tp_dict */
    0, /* tp_descr_get */
    0, /* tp_descr_set */
    0, /* tp_dictoffset */
    (initproc)Noddy_init, /* tp_init */
    0, /* tp_alloc */
    Noddy_new, /* tp_new */
};

static PyMethodDef module_methods[] = {
    {NULL} /* Sentinel */
};
To provide greater control, over the first and last attributes, we'll use custom getter and setter functions. Here are the functions for getting and setting the first attribute:

```c
Noddy_getfirst(Noddy *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Noddy_setfirst(Noddy *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }

    if (!PyString_Check(value)) {
        PyErr_SetString(PyExc_TypeError, "The first attribute value must be a string");
        return -1;
    }
```
PyDECREF(self->first);
Py_INCREF(value);
self->first = value;

return 0;
}

The getter function is passed a Noddy object and a “closure”, which is void pointer. In this case, the closure is ignored. (The closure supports an advanced usage in which definition data is passed to the getter and setter. This could, for example, be used to allow a single set of getter and setter functions that decide the attribute to get or set based on data in the closure.)

The setter function is passed the Noddy object, the new value, and the closure. The new value may be NULL, in which case the attribute is being deleted. In our setter, we raise an error if the attribute is deleted or if the attribute value is not a string.

We create an array of PyGetSetDef structures:

```c
static PyGetSetDef Noddy_getseters[] = {
    {"first",
     (getter)Noddy_getfirst, (setter)Noddy_setfirst,
     "first name",
     NULL},
    {"last",
     (getter)Noddy_getlast, (setter)Noddy_setlast,
     "last name",
     NULL},
    {NULL} /* Sentinel */
};
```

and register it in the tp_getset slot:

```c
Noddy_getseters, /* tp_getset */
```

to register our attribute getters and setters.
The last item in a `PyGetSetDef` structure is the closure mentioned above. In this case, we aren’t using the closure, so we just pass `NULL`.

We also remove the member definitions for these attributes:

```c
static PyMemberDef Noddy_members[] = {
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL} /* Sentinel */
};
```

We also need to update the `tp_init` handler to only allow strings [3] to be passed:

```c
static int Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|SSi", kwlist
            &first, &last,
            &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }

    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }

    return 0;
}
```
With these changes, we can assure that the first and last members are never NULL so we can remove checks for NULL values in almost all cases. This means that most of the Py_XDECREF() calls can be converted to Py_DECREF() calls. The only place we can’t change these calls is in the deallocator, where there is the possibility that the initialization of these members failed in the constructor.

We also rename the module initialization function and module name in the initialization function, as we did before, and we add an extra definition to the setup.py file.

### 2.1.3. Supporting cyclic garbage collection

Python has a cyclic-garbage collector that can identify unneeded objects even when their reference counts are not zero. This can happen when objects are involved in cycles. For example, consider:

```python
>>> l = []
>>> l.append(l)
>>> del l
```

In this example, we create a list that contains itself. When we delete it, it still has a reference from itself. Its reference count doesn’t drop to zero. Fortunately, Python’s cyclic-garbage collector will eventually figure out that the list is garbage and free it.

In the second version of the Noddy example, we allowed any kind of object to be stored in the first or last attributes. [4] This means that Noddy objects can participate in cycles:

```python
>>> import noddy2
>>> n = noddy2.Noddy()
>>> l = [n]
>>> n.first = l
```
This is pretty silly, but it gives us an excuse to add support for the cyclic-garbage collector to the **Noddy** example. To support cyclic garbage collection, types need to fill two slots and set a class flag that enables these slots:

```c
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    PyObject *first;
    PyObject *last;
    int number;
} Noddy;

static int
Noddy_traverse(Noddy *self, visitproc visit, void *arg)
{
    int vret;

    if (self->first) {
        vret = visit(self->first, arg);
        if (vret != 0)
            return vret;
    }

    if (self->last) {
        vret = visit(self->last, arg);
        if (vret != 0)
            return vret;
    }

    return 0;
}

static int
Noddy_clear(Noddy *self)
{
    PyObject *tmp;

    tmp = self->first;
    self->first = NULL;
    Py_XDECREF(tmp);

    tmp = self->last;
    self->last = NULL;
```
Py_XDECREF(
tmp);

return 0;
}

static void
Noddy_dealloc(Noddy* self)
{
    Noddy_clear(self);
    self->ob_type->tp_free((PyObject*)self);
}

static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL)
        {
            Py_DECREF(self);
            return NULL;
        }
    }

    self->last = PyString_FromString("");
    if (self->last == NULL)
    {
        Py_DECREF(self);
        return NULL;
    }

    self->number = 0;
}

return (PyObject *)self;
}

static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", kwlist,

return -1;

if (first) {
    tmp = self->first;
    Py_INCREF(first);
    self->first = first;
    Py_XDECREF(tmp);
}

if (last) {
    tmp = self->last;
    Py_INCREF(last);
    self->last = last;
    Py_XDECREF(tmp);
}

return 0;

static PyMemberDef Noddy_members[] = {
    {"first", T_OBJECT_EX, offsetof(Noddy, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(Noddy, last), 0,
     "last name"},
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL} /* Sentinel */
};

static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
        format = PyString_FromString("%s %s");
        if (format == NULL)
            return NULL;
    }

    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }

    return NULL;
}
if (self->last == NULL) {
    PyErr_SetString(PyExc_AttributeError, "last");
    return NULL;
}

args = Py_BuildValue("OO", self->first, self->last);
if (args == NULL)
    return NULL;

result = PyString_Format(format, args);
Py_DECREF(args);

return result;

static PyMethodDef Noddy_methods[] = {
    {"name",  (PyCFunction)Noddy_name, METH_NOARGS,
        "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};

static PyTypeObject NoddyType = {
    PyObject_HEAD_INIT(NULL)
    0,               /*ob_size*/
    "noddy.Noddy",  /*tp_name*/
    sizeof(Noddy),  /*tp_basicsize*/
    0,               /*tp_itemsize*/
    (destructor)Noddy_dealloc, /*tp_dealloc*/
    0,               /*tp_print*/
    0,               /*tp_getattr*/
    0,               /*tp_setattr*/
    0,               /*tp_compare*/
    0,               /*tp_repr*/
    0,               /*tp_as_number*/
    0,               /*tp_as_sequence*/
    0,               /*tp_as_mapping*/
    0,               /*tp_hash */
    0,               /*tp_call*/
    0,               /*tp_str*/
    0,               /*tp_getattro*/
    0,               /*tp_setattro*/
    0,               /*tp_as_buffer*/
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
    "Noddy objects",     /* tp_doc */
};
The traversal method provides access to subobjects that could
participate in cycles:

```c
static int Noddy_traverse(Noddy *self, visitproc visit, void *arg)
{
    int vret;

    if (self->first) {
        vret = visit(self->first, arg);
        if (vret != 0)
            return vret;
    }
    if (self->last) {
        vret = visit(self->last, arg);
        if (vret != 0)
            return vret;
    }
    return 0;
}
```

For each subobject that can participate in cycles, we need to call the `visit()` function, which is passed to the traversal method. The `visit()` function takes as arguments the subobject and the extra argument `arg` passed to the traversal method. It returns an integer value that must be returned if it is non-zero.

Python 2.4 and higher provide a `Py_VISIT()` macro that automates calling visit functions. With `Py_VISIT()`, `Noddy_traverse()` can be simplified:

```c
static int Noddy_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->first);
    Py_VISIT(self->last);
    return 0;
}
```

**Note:** Note that the `tp_traverse` implementation must name its
arguments exactly visit and arg in order to use Py_VISIT(). This is to encourage uniformity across these boring implementations.

We also need to provide a method for clearing any subobjects that can participate in cycles. We implement the method and reimplement the deallocator to use it:

```c
static int
Noddy_clear(Noddy *self)
{
    PyObject *tmp;

    tmp = self->first;
    self->first = NULL;
    Py_XDECREF(tmp);

    tmp = self->last;
    self->last = NULL;
    Py_XDECREF(tmp);

    return 0;
}

static void
Noddy_dealloc(Noddy *self)
{
    Noddy_clear(self);
    self->ob_type->tp_free((PyObject*)self);
}
```

Notice the use of a temporary variable in Noddy_clear(). We use the temporary variable so that we can set each member to NULL before decrementing its reference count. We do this because, as was discussed earlier, if the reference count drops to zero, we might cause code to run that calls back into the object. In addition, because we now support garbage collection, we also have to worry about code being run that triggers garbage collection. If garbage collection is run, our tp_traverse handler could get called. We can’t take a chance of having Noddy_traverse() called when a member’s reference count has dropped to zero and its value hasn’t been set to
NULL.

Python 2.4 and higher provide a `Py_CLEAR()` that automates the careful decrementing of reference counts. With `Py_CLEAR()`, the `Noddy_clear()` function can be simplified:

```c
static int
Noddy_clear(Noddy *self)
{
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}
```

Finally, we add the `Py_TPFLAGS_HAVE_GC` flag to the class flags:

```
Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
```

That's pretty much it. If we had written custom `tp_alloc` or `tp_free` slots, we'd need to modify them for cyclic-garbage collection. Most extensions will use the versions automatically provided.

2.1.4. Subclassing other types

It is possible to create new extension types that are derived from existing types. It is easiest to inherit from the built-in types, since an extension can easily use the `PyTypeObject` it needs. It can be difficult to share these `PyTypeObject` structures between extension modules.

In this example we will create a `Shoddy` type that inherits from the built-in `list` type. The new type will be completely compatible with regular lists, but will have an additional `increment()` method that increases an internal counter.

```python
>>> import shoddy
```
```python
>>> s = shoddy.Shoddy(range(3))
>>> s.extend(s)
>>> print len(s)
6
>>> print s.increment()
1
>>> print s.increment()
2
```

```c
#include <Python.h>

typedef struct {
  PyListObject list;
  int state;
} Shoddy;

static PyObject *
Shoddy_increment(Shoddy *self, PyObject *unused)
{
  self->state++;
  return PyInt_FromLong(self->state);
}

static PyMethodDef Shoddy_methods[] = {
  {"increment", (PyCFunction)shoddy_increment, METH_NOARGS,
   PyDoc_STR("increment state counter")},
  {NULL, NULL},
};

static int
Shoddy_init(Shoddy *self, PyObject *args, PyObject *kwds)
{
  if (PyList_Type.tp_init((PyObject *)self, args, kwds) < 0)
    return -1;
  self->state = 0;
  return 0;
}

static PyTypeObject ShoddyType = {
  PyObject_HEAD_INIT(NULL)
  0, /* ob_size */
  "shoddy.Shoddy", /* tp_name */
  sizeof(Shoddy), /* tp_basicsize */
```
ShoddyType.tp_base = &PyList_Type;

PyMODINIT_FUNC
initshoddy(void)
{
    PyObject *m;

    ShoddyType.tp_base = &PyList_Type;
    if (PyType_Ready(&ShoddyType) < 0)
        return;
}
As you can see, the source code closely resembles the Noddy examples in previous sections. We will break down the main differences between them.

define struct {
    PyListObject list;
    int state;
} Shoddy;

The primary difference for derived type objects is that the base type's object structure must be the first value. The base type will already include the `PyObject_HEAD()` at the beginning of its structure.

When a Python object is a Shoddy instance, its PyObject* pointer can be safely cast to both PyListObject* and Shoddy*.

```c
static int Shoddy_init(Shoddy *self, PyObject *args, PyObject *kwds) {
    if (PyList_Type.tp_init((PyObject *)self, args, kwds) < 0)
        return -1;
    self->state = 0;
    return 0;
}
```

In the `__init__` method for our type, we can see how to call through to the `__init__` method of the base type.

This pattern is important when writing a type with custom `new` and `dealloc` methods. The `new` method should not actually create the memory for the object with `tp_alloc`, that will be handled by the base
class when calling its \texttt{tp\_new}.

When filling out the \texttt{PyTypeObject()} for the \texttt{Shoddy} type, you see a slot for \texttt{tp\_base()}. Due to cross platform compiler issues, you can’t fill that field directly with the \texttt{PyList\_Type()}; it can be done later in the module’s \texttt{init()} function.

```python
PyMODINIT_FUNC
initshoddy(void)
{
    PyObject *m;

    ShoddyType.tp_base = &PyList_Type;
    if (PyType_Ready(&ShoddyType) < 0)
        return;

    m = Py_InitModule3("shoddy", NULL, "Shoddy module");
    if (m == NULL)
        return;

    Py_INCREF(&ShoddyType);
    PyModule_AddObject(m, "Shoddy", (PyObject *) &ShoddyType);
}
```

Before calling \texttt{PyType\_Ready()}, the type structure must have the \texttt{tp\_base} slot filled in. When we are deriving a new type, it is not necessary to fill out the \texttt{tp\_alloc} slot with \texttt{PyType\_Generic\_New()} – the allocate function from the base type will be inherited.

After that, calling \texttt{PyType\_Ready()} and adding the type object to the module is the same as with the basic \texttt{Noddy} examples.
This section aims to give a quick fly-by on the various type methods you can implement and what they do.

Here is the definition of `PyTypeObject`, with some fields only used in debug builds omitted:

```c
typedef struct _typeobject {
    PyObject_VAR_HEAD
    char *tp_name; /* For printing, in format "<module>.<name>" */
    int tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

destructor tp_dealloc;
printfunc tp_print;
getattrofunc tp_getattr;
setattrofunc tp_setattr;
cmpfunc tp_compare;
reprfunc tp_repr;

    /* Method suites for standard classes */

PyNumberMethods *tp_as_number;
 PySequenceMethods *tp_as_sequence;
 PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

hashfunc tp_hash;
ternaryfunc tp_call;
reprfunc tp_str;
geattrofunc tp_getattro;
setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
long tp_flags;
```
char *tp_doc; /* Documentation string */

/* Assigned meaning in release 2.0 */
/* call function for all accessible objects */
traverseproc tp_traverse;

/* delete references to contained objects */
inquiry tp_clear;

/* Assigned meaning in release 2.1 */
/* rich comparisons */
richcmpfunc tp_richcompare;

/* weak reference enabler */
long tp_weaklistoffset;

/* Added in release 2.2 */
/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;

/* Attribute descriptor and subclassing stuff */
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct _typeobject *tp_base;
PyObject *tp_dict;
descrgetfunc tp_descr_get;
descrsetfunc tp_descr_set;
long tp_dictoffset;
initproc tp_init;
allocfunc tp_alloc;
newfunc tp_new;
freefunc tp_free; /* Low-level free-memory routine */
inquiry tp_is_gc; /* For PyObject_IS_GC */
PyObject *tp_bases;
PyObject *tp_mro; /* method resolution order */
PyObject *tp_cache;
PyObject *tp_subclasses;
PyObject *tp_weaklist;

} PyTypeObject;

Now that’s a lot of methods. Don’t worry too much though - if you have a type you want to define, the chances are very good that you
will only implement a handful of these.

As you probably expect by now, we're going to go over this and give more information about the various handlers. We won't go in the order they are defined in the structure, because there is a lot of historical baggage that impacts the ordering of the fields; be sure your type initialization keeps the fields in the right order! It's often easiest to find an example that includes all the fields you need (even if they're initialized to 0) and then change the values to suit your new type.

```c
char *tp_name; /* For printing */
```

The name of the type - as mentioned in the last section, this will appear in various places, almost entirely for diagnostic purposes. Try to choose something that will be helpful in such a situation!

```c
int tp_basicsize, tp_itemsize; /* For allocation */
```

These fields tell the runtime how much memory to allocate when new objects of this type are created. Python has some built-in support for variable length structures (think: strings, lists) which is where the `tp_itemsize` field comes in. This will be dealt with later.

```c
char *tp_doc;
```

Here you can put a string (or its address) that you want returned when the Python script references `obj.__doc__` to retrieve the doc string.

Now we come to the basic type methods—the ones most extension types will implement.

### 2.2.1. Finalization and De-allocation
This function is called when the reference count of the instance of your type is reduced to zero and the Python interpreter wants to reclaim it. If your type has memory to free or other clean-up to perform, put it here. The object itself needs to be freed here as well. Here is an example of this function:

```c
static void
newdatatype_dealloc(newdatatypeobject * obj)
{
    free(obj->obj_UnderlyingDatatypePtr);
    obj->ob_type->tp_free(obj);
}
```

One important requirement of the deallocator function is that it leaves any pending exceptions alone. This is important since deallocators are frequently called as the interpreter unwinds the Python stack; when the stack is unwound due to an exception (rather than normal returns), nothing is done to protect the deallocators from seeing that an exception has already been set. Any actions which a deallocator performs which may cause additional Python code to be executed may detect that an exception has been set. This can lead to misleading errors from the interpreter. The proper way to protect against this is to save a pending exception before performing the unsafe action, and restoring it when done. This can be done using the `PyErr_Fetch()` and `PyErr_Restore()` functions:

```c
static void
my_dealloc(PyObject *obj)
{
    MyObject *self = (MyObject *) obj;
    PyObject *cbresult;

    if (self->my_callback != NULL) {
        PyObject *err_type, *err_value, *err_traceback;
        int have_error = PyErr_Occurred() ? 1 : 0;

        if (have_error) 
```
2.2.2. Object Presentation

In Python, there are three ways to generate a textual representation of an object: the `repr()` function (or equivalent back-tick syntax), the `str()` function, and the `print` statement. For most objects, the `print` statement is equivalent to the `str()` function, but it is possible to special-case printing to a `FILE*` if necessary; this should only be done if efficiency is identified as a problem and profiling suggests that creating a temporary string object to be written to a file is too expensive.

These handlers are all optional, and most types at most need to implement the `tp_str` and `tp_repr` handlers.

```c
reprfunc tp_repr;
reprfunc tp_str;
printfunc tp_print;
```

The `tp_repr` handler should return a string object containing a representation of the instance for which it is called. Here is a simple example:
static PyObject *
newdatatype_repr(newdatatypeobject * obj)
{
    return PyString_FromFormat("Repr-ified_newdatatype{{size:\%d}}", obj->obj_UnderlyingDatatypePtr->)
}

If no \texttt{tp\_repr} handler is specified, the interpreter will supply a representation that uses the type's \texttt{tp\_name} and a uniquely-identifying value for the object.

The \texttt{tp\_str} handler is to \texttt{str()} what the \texttt{tp\_repr} handler described above is to \texttt{repr()}; that is, it is called when Python code calls \texttt{str()} on an instance of your object. Its implementation is very similar to the \texttt{tp\_repr} function, but the resulting string is intended for human consumption. If \texttt{tp\_str} is not specified, the \texttt{tp\_repr} handler is used instead.

Here is a simple example:

static PyObject *
newdatatype_str(newdatatypeobject * obj)
{
    return PyString_FromFormat("Stringified_newdatatype{{size:\%d}}", obj->obj_UnderlyingDatatypePtr->)
}

The print function will be called whenever Python needs to “print” an instance of the type. For example, if ‘node’ is an instance of type TreeNode, then the print function is called when Python code calls:

```
print node
```

There is a flags argument and one flag, \texttt{PY\_PRINT\_RAW}, and it suggests that you print without string quotes and possibly without interpreting escape sequences.
The print function receives a file object as an argument. You will likely want to write to that file object.

Here is a sample print function:

```c
static int
newdatatype_print(newdatatypeobject *obj, FILE *fp, int flags)
{
    if (flags & Py_PRINT_RAW) {
        fprintf(fp, "<\{newdatatype object--size: %d}\>",
                obj->obj_UnderlyingDatatypePtr->size);
    } else {
        fprintf(fp, ""<\{newdatatype object--size: %d}\>\"",
                obj->obj_UnderlyingDatatypePtr->size);
    }
    return 0;
}
```

### 2.2.3. Attribute Management

For every object which can support attributes, the corresponding type must provide the functions that control how the attributes are resolved. There needs to be a function which can retrieve attributes (if any are defined), and another to set attributes (if setting attributes is allowed). Removing an attribute is a special case, for which the new value passed to the handler is `NULL`.

Python supports two pairs of attribute handlers; a type that supports attributes only needs to implement the functions for one pair. The difference is that one pair takes the name of the attribute as a `char*`, while the other accepts a `PyObject*`. Each type can use whichever pair makes more sense for the implementation's convenience.

```c
getattrfunc tp_getattr; /* char * version */
setattrfunc tp_setattr;
/* ... */
getattrofunc tp_getattrofunc; /* PyObject * version */
setattrofunc tp_setattrofunc;
```
If accessing attributes of an object is always a simple operation (this will be explained shortly), there are generic implementations which can be used to provide the PyObject* version of the attribute management functions. The actual need for type-specific attribute handlers almost completely disappeared starting with Python 2.2, though there are many examples which have not been updated to use some of the new generic mechanism that is available.

### 2.2.3.1. Generic Attribute Management

*New in version 2.2.*

Most extension types only use *simple* attributes. So, what makes the attributes simple? There are only a couple of conditions that must be met:

1. The name of the attributes must be known when `PyType_Ready()` is called.
2. No special processing is needed to record that an attribute was looked up or set, nor do actions need to be taken based on the value.

Note that this list does not place any restrictions on the values of the attributes, when the values are computed, or how relevant data is stored.

When `PyType_Ready()` is called, it uses three tables referenced by the type object to create *descriptors* which are placed in the dictionary of the type object. Each descriptor controls access to one attribute of the instance object. Each of the tables is optional; if all three are `NULL`, instances of the type will only have attributes that are inherited from their base type, and should leave the `tp_getattro` and `tp_setattro` fields `NULL` as well, allowing the base type to handle attributes.
The tables are declared as three fields of the type object:

```c
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
```

If `tp_methods` is not `NULL`, it must refer to an array of `PyMethodDef` structures. Each entry in the table is an instance of this structure:

```c
typedef struct PyMethodDef {
    const char *ml_name; /* method name */
    PyCFunction ml_meth; /* implementation function */
    int ml_flags; /* flags */
    const char *ml_doc; /* docstring */
} PyMethodDef;
```

One entry should be defined for each method provided by the type; no entries are needed for methods inherited from a base type. One additional entry is needed at the end; it is a sentinel that marks the end of the array. The `ml_name` field of the sentinel must be `NULL`.

XXX Need to refer to some unified discussion of the structure fields, shared with the next section.

The second table is used to define attributes which map directly to data stored in the instance. A variety of primitive C types are supported, and access may be read-only or read-write. The structures in the table are defined as:

```c
typedef struct PyMemberDef {
    char *name;
    int type;
    int offset;
    int flags;
    char *doc;
} PyMemberDef;
```

For each entry in the table, a descriptor will be constructed and added to the type which will be able to extract a value from the
instance structure. The type field should contain one of the type codes defined in the structmember.h header; the value will be used to determine how to convert Python values to and from C values. The flags field is used to store flags which control how the attribute can be accessed.

XXX Need to move some of this to a shared section!

The following flag constants are defined in structmember.h; they may be combined using bitwise-OR.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>READONLY</td>
<td>Never writable.</td>
</tr>
<tr>
<td>RO</td>
<td>Shorthand for READONLY.</td>
</tr>
<tr>
<td>READ_RESTRICTED</td>
<td>Not readable in restricted mode.</td>
</tr>
<tr>
<td>WRITE_RESTRICTED</td>
<td>Not writable in restricted mode.</td>
</tr>
<tr>
<td>RESTRICTED</td>
<td>Not readable or writable in restricted mode.</td>
</tr>
</tbody>
</table>

An interesting advantage of using the tp_members table to build descriptors that are used at runtime is that any attribute defined this way can have an associated doc string simply by providing the text in the table. An application can use the introspection API to retrieve the descriptor from the class object, and get the doc string using its __doc__ attribute.

As with the tp_methods table, a sentinel entry with a name value of NULL is required.

2.2.3.2. Type-specific Attribute Management

For simplicity, only the char* version will be demonstrated here; the type of the name parameter is the only difference between the char* and PyObject* flavors of the interface. This example effectively does
the same thing as the generic example above, but does not use the generic support added in Python 2.2. The value in showing this is two-fold: it demonstrates how basic attribute management can be done in a way that is portable to older versions of Python, and explains how the handler functions are called, so that if you do need to extend their functionality, you'll understand what needs to be done.

The `tp_getattr` handler is called when the object requires an attribute look-up. It is called in the same situations where the `__getattr__()` method of a class would be called.

A likely way to handle this is (1) to implement a set of functions (such as `newdatatype_getSize()`, and `newdatatype_setSize()` in the example below), (2) provide a method table listing these functions, and (3) provide a getattr function that returns the result of a lookup in that table. The method table uses the same structure as the `tp_methods` field of the type object.

Here is an example:

```c
static PyMethodDef newdatatype_methods[] = {
    {"getSize", (PyCFunction)newdatatype_getSize, METH_VARARGS,
     "Return the current size."},
    {"setSize", (PyCFunction)newdatatype_setSize, METH_VARARGS,
     "Set the size."},
    {NULL, NULL, 0, NULL} /* sentinel */
};

static PyObject *
newdatatype_getattr(newdatatypeobject *obj, char *name)
{
    return Py_FindMethod(newdatatype_methods, (PyObject *)obj,
}
```

The `tp_setattr` handler is called when the `__setattr__()`, `__delattr__()`, or `__setattr__()`, or `__delattr__() `method of a class instance would be called. When an
attribute should be deleted, the third parameter will be NULL. Here is an example that simply raises an exception; if this were really all you wanted, the `tp_setattr` handler should be set to `NULL`.

```c
static int
newdatatype_setattr(newdatatypeobject *obj, char *name, PyObject *)
{
    (void)PyErr_Format(PyExc_RuntimeError, "Read-only attribute
return -1;
}
```

### 2.2.4. Object Comparison

```c
cmpfunc tp_compare;
```

The `tp_compare` handler is called when comparisons are needed and the object does not implement the specific rich comparison method which matches the requested comparison. (It is always used if defined and the `PyObject_Compare()` or `PyObject_Cmp()` functions are used, or if `cmp()` is used from Python.) It is analogous to the `__cmp__()` method. This function should return `-1` if `obj1` is less than `obj2`, `0` if they are equal, and `1` if `obj1` is greater than `obj2`. (It was previously allowed to return arbitrary negative or positive integers for less than and greater than, respectively; as of Python 2.2, this is no longer allowed. In the future, other return values may be assigned a different meaning.)

A `tp_compare` handler may raise an exception. In this case it should return a negative value. The caller has to test for the exception using `PyErr_Occurred()`.

Here is a sample implementation:

```c
static int
newdatatype_compare(newdatatypeobject * obj1, newdatatypeobject
```
2.2.5. Abstract Protocol Support

Python supports a variety of abstract ‘protocols;‘ the specific interfaces provided to use these interfaces are documented in Abstract Objects Layer.

A number of these abstract interfaces were defined early in the development of the Python implementation. In particular, the number, mapping, and sequence protocols have been part of Python since the beginning. Other protocols have been added over time. For protocols which depend on several handler routines from the type implementation, the older protocols have been defined as optional blocks of handlers referenced by the type object. For newer protocols there are additional slots in the main type object, with a flag bit being set to indicate that the slots are present and should be checked by the interpreter. (The flag bit does not indicate that the slot values are non-NULL. The flag may be set to indicate the presence of a slot, but a slot may still be unfilled.)

```c
PyNumberMethods       *tp_as_number;
PySequenceMethods     *tp_as_sequence;
PyMappingMethods      *tp_as_mapping;
```
If you wish your object to be able to act like a number, a sequence, or a mapping object, then you place the address of a structure that implements the C type `PyNumberMethods`, `PySequenceMethods`, or `PyMappingMethods`, respectively. It is up to you to fill in this structure with appropriate values. You can find examples of the use of each of these in the `objects` directory of the Python source distribution.

```c
hashfunc tp_hash;
```

This function, if you choose to provide it, should return a hash number for an instance of your data type. Here is a moderately pointless example:

```c
static long
newdatatype_hash(newdatatypeobject *obj)
{
    long result;
    result = obj->obj_UnderlyingDatatypePtr->size;
    result = result * 3;
    return result;
}
```

```c
ternaryfunc tp_call;
```

This function is called when an instance of your data type is “called”, for example, if `obj1` is an instance of your data type and the Python script contains `obj1('hello')`, the `tp_call` handler is invoked.

This function takes three arguments:

1. `arg1` is the instance of the data type which is the subject of the call. If the call is `obj1('hello')`, then `arg1` is `obj1`.
2. `arg2` is a tuple containing the arguments to the call. You can use `PyArg_ParseTuple()` to extract the arguments.
3. `arg3` is a dictionary of keyword arguments that were passed. If
this is non-NULL and you support keyword arguments, use\nPyArg_ParseTupleAndKeywords() to extract the arguments. If you\ndo not want to support keyword arguments and this is non-\nNULL, raise a TypeError with a message saying that keyword\narguments are not supported.

Here is a desultory example of the implementation of the call\nfunction.

```c
/* Implement the call function.\n *   obj1 is the instance receiving the call.\n *   obj2 is a tuple containing the arguments to the call, in\n *   case 3 strings.\n */
static PyObject *\nnewdatatype_call(newdatatypeobject *obj, PyObject *args, PyObje\n{
    PyObject *result;
    char *arg1;
    char *arg2;
    char *arg3;

    if (!PyArg_ParseTuple(args, "sss:call", &arg1, &arg2, &arg3\n        return NULL;
    }
    result = PyString_FromFormat(\n        "Returning -- value: [%d] arg1: [%s] arg2: [%s] arg3\n    obj->obj_UnderlyingDatatypePtr->size,\n    arg1, arg2, arg3);
    printf("%s", PyString_AS_STRING(result));\n    return result;
}
```

XXX some fields need to be added here...

```c
/* Added in release 2.2 */
/* Iterators */
getiterfunc tp_iter;
ithernextfunc tp_iternext;
```

These functions provide support for the iterator protocol. Any object
which wishes to support iteration over its contents (which may be generated during iteration) must implement the \texttt{tp\_iter} handler. Objects which are returned by a \texttt{tp\_iter} handler must implement both the \texttt{tp\_iter} and \texttt{tp\_iternext} handlers. Both handlers take exactly one parameter, the instance for which they are being called, and return a new reference. In the case of an error, they should set an exception and return \texttt{NULL}.

For an object which represents an iterable collection, the \texttt{tp\_iter} handler must return an iterator object. The iterator object is responsible for maintaining the state of the iteration. For collections which can support multiple iterators which do not interfere with each other (as lists and tuples do), a new iterator should be created and returned. Objects which can only be iterated over once (usually due to side effects of iteration) should implement this handler by returning a new reference to themselves, and should also implement the \texttt{tp\_iternext} handler. File objects are an example of such an iterator.

Iterator objects should implement both handlers. The \texttt{tp\_iter} handler should return a new reference to the iterator (this is the same as the \texttt{tp\_iter} handler for objects which can only be iterated over destructively). The \texttt{tp\_iternext} handler should return a new reference to the next object in the iteration if there is one. If the iteration has reached the end, it may return \texttt{NULL} without setting an exception or it may set \texttt{StopIteration}; avoiding the exception can yield slightly better performance. If an actual error occurs, it should set an exception and return \texttt{NULL}.

\section*{2.2.6. Weak Reference Support}

One of the goals of Python's weak-reference implementation is to allow any type to participate in the weak reference mechanism
without incurring the overhead on those objects which do not benefit by weak referencing (such as numbers).

For an object to be weakly referencable, the extension must include a `PyObject*` field in the instance structure for the use of the weak reference mechanism; it must be initialized to `NULL` by the object’s constructor. It must also set the `tp_weaklistoffset` field of the corresponding type object to the offset of the field. For example, the instance type is defined with the following structure:

```c
typedef struct {
    PyObject_HEAD
    PyClassObject  *in_class;  /* The class object */
    PyObject       *in_dict;   /* A dictionary */
    PyObject       *in_weakreflist; /* List of weak references */
} PyInstanceObject;
```

The statically-declared type object for instances is defined this way:

```c
PyTypeObject PyInstance_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,
    "module.instance",
    /* Lots of stuff omitted for brevity... */
    Py_TPFLAGS_DEFAULT,
    0,
    0,
    0,
    0,
    0,
    offsetof(PyInstanceObject, in_weakreflist), /* tp_weaklistoffset */
};
```

The type constructor is responsible for initializing the weak reference list to `NULL`:

```c
static PyObject *
instance_new() {
```
The only further addition is that the destructor needs to call the weak reference manager to clear any weak references. This is only required if the weak reference list is non-NULL:

```c
static void instance_dealloc(PyInstanceObject *inst)
{
    /* Allocate temporaries if needed, but do not begin destruction just yet. */
    if (inst->in_weakreflist != NULL)
        PyObject_ClearWeakRefs((PyObject *) inst);
    /* Proceed with object destruction normally. */
}
```

### 2.2.7. More Suggestions

Remember that you can omit most of these functions, in which case you provide 0 as a value. There are type definitions for each of the functions you must provide. They are in `object.h` in the Python include directory that comes with the source distribution of Python.

In order to learn how to implement any specific method for your new data type, do the following: Download and unpack the Python source distribution. Go to the `objects` directory, then search the C source files for `tp_` plus the function you want (for example, `tp_print` or `tp_compare`). You will find examples of the function you want to implement.

When you need to verify that an object is an instance of the type you
are implementing, use the `PyObject_TypeCheck()` function. A sample of its use might be something like the following:

```c
if (!PyObject_TypeCheck(some_object, &MyType)) {
    PyErr_SetString(PyExc_TypeError, "arg #1 not a mything");
    return NULL;
}
```

Footnotes

[1] This is true when we know that the object is a basic type, like a string or a float.
   We relied on this in the `tp_dealloc` handler in this example, because our type doesn't support garbage collection. Even if a type supports garbage collection, there are calls that can be made to “untrack” the object from garbage collection, however, these calls are advanced and not covered here.

We now know that the first and last members are strings, so perhaps we could be less careful about decrementing their reference counts, however, we accept instances of string subclasses. Even though deallocating normal strings won’t call back into our objects, we can’t guarantee that deallocating an instance of a string subclass won’t call back into our objects.

Even in the third version, we aren’t guaranteed to avoid cycles.

[3] Instances of string subclasses are allowed and string subclasses could allow cycles even if normal strings don’t.
3. Building C and C++ Extensions with distutils

Starting in Python 1.4, Python provides, on Unix, a special make file for building make files for building dynamically-linked extensions and custom interpreters. Starting with Python 2.0, this mechanism (known as related to Makefile.pre.in, and Setup files) is no longer supported. Building custom interpreters was rarely used, and extension modules can be built using distutils.

Building an extension module using distutils requires that distutils is installed on the build machine, which is included in Python 2.x and available separately for Python 1.5. Since distutils also supports creation of binary packages, users don’t necessarily need a compiler and distutils to install the extension.

A distutils package contains a driver script, setup.py. This is a plain Python file, which, in the most simple case, could look like this:

```python
from distutils.core import setup, Extension

module1 = Extension('demo',
                     sources = ['demo.c'])

setup (name = 'PackageName',
        version = '1.0',
        description = 'This is a demo package',
        ext_modules = [module1])
```

With this setup.py, and a file demo.c, running

```bash
python setup.py build
```

will compile demo.c, and produce an extension module named demo in the build directory. Depending on the system, the module file will
end up in a subdirectory `build/lib.system`, and may have a name like `demo.so` or `demo.pyd`.

In the `setup.py`, all execution is performed by calling the `setup` function. This takes a variable number of keyword arguments, of which the example above uses only a subset. Specifically, the example specifies meta-information to build packages, and it specifies the contents of the package. Normally, a package will contain of addition modules, like Python source modules, documentation, subpackages, etc. Please refer to the distutils documentation in *Distributing Python Modules (Legacy version)* to learn more about the features of distutils; this section explains building extension modules only.

It is common to pre-compute arguments to `setup()`, to better structure the driver script. In the example above, the `ext_modules` argument to `setup()` is a list of extension modules, each of which is an instance of the `Extension`. In the example, the instance defines an extension named `demo` which is build by compiling a single source file, `demo.c`.

In many cases, building an extension is more complex, since additional preprocessor defines and libraries may be needed. This is demonstrated in the example below.

```python
from distutils.core import setup, Extension

module1 = Extension('demo',
    define_macros = [('MAJOR_VERSION', '1'), ('MINOR_VERSION', '0')],
    include_dirs = ['/usr/local/include'],
    libraries = ['tcl83'],
    library_dirs = ['/usr/local/lib'],
    sources = ['demo.c'])

setup (name = 'PackageName',
        version = '1.0',
        ...)
```
In this example, `setup()` is called with additional meta-information, which is recommended when distribution packages have to be built. For the extension itself, it specifies preprocessor defines, include directories, library directories, and libraries. Depending on the compiler, distutils passes this information in different ways to the compiler. For example, on Unix, this may result in the compilation commands:

```bash
gcc -DNDEBUG -g -03 -Wall -Wstrict-prototypes -fPIC -DMAJOR_VERSION
```

```bash
gcc -shared build/temp.linux-i686-2.2/demo.o -L/usr/local/lib
```

These lines are for demonstration purposes only; distutils users should trust that distutils gets the invocations right.
3.1. Distributing your extension modules

When an extension has been successfully build, there are three ways to use it.

End-users will typically want to install the module, they do so by running

```
python setup.py install
```

Module maintainers should produce source packages; to do so, they run

```
python setup.py sdist
```

In some cases, additional files need to be included in a source distribution; this is done through a `MANIFEST.in` file; see the distutils documentation for details.

If the source distribution has been build successfully, maintainers can also create binary distributions. Depending on the platform, one of the following commands can be used to do so.

```
python setup.py bdist_wininst
python setup.py bdist_rpm
python setup.py bdist_dumb
```
This chapter briefly explains how to create a Windows extension module for Python using Microsoft Visual C++, and follows with more detailed background information on how it works. The explanatory material is useful for both the Windows programmer learning to build Python extensions and the Unix programmer interested in producing software which can be successfully built on both Unix and Windows.

Module authors are encouraged to use the distutils approach for building extension modules, instead of the one described in this section. You will still need the C compiler that was used to build Python; typically Microsoft Visual C++.

**Note:** This chapter mentions a number of filenames that include an encoded Python version number. These filenames are represented with the version number shown as XY; in practice, 'X' will be the major version number and 'Y' will be the minor version number of the Python release you’re working with. For example, if you are using Python 2.2.1, XY will actually be 22.
4.1. A Cookbook Approach

There are two approaches to building extension modules on Windows, just as there are on Unix: use the distutils package to control the build process, or do things manually. The distutils approach works well for most extensions; documentation on using distutils to build and package extension modules is available in Distributing Python Modules (Legacy version). If you find you really need to do things manually, it may be instructive to study the project file for the winsound standard library module.
4.2. Differences Between Unix and Windows

Unix and Windows use completely different paradigms for run-time loading of code. Before you try to build a module that can be dynamically loaded, be aware of how your system works.

In Unix, a shared object (.so) file contains code to be used by the program, and also the names of functions and data that it expects to find in the program. When the file is joined to the program, all references to those functions and data in the file's code are changed to point to the actual locations in the program where the functions and data are placed in memory. This is basically a link operation.

In Windows, a dynamic-link library (.dll) file has no dangling references. Instead, an access to functions or data goes through a lookup table. So the DLL code does not have to be fixed up at runtime to refer to the program’s memory; instead, the code already uses the DLL's lookup table, and the lookup table is modified at runtime to point to the functions and data.

In Unix, there is only one type of library file (.a) which contains code from several object files (.o). During the link step to create a shared object file (.so), the linker may find that it doesn’t know where an identifier is defined. The linker will look for it in the object files in the libraries; if it finds it, it will include all the code from that object file.

In Windows, there are two types of library, a static library and an import library (both called .lib). A static library is like a Unix .a file; it contains code to be included as necessary. An import library is basically used only to reassure the linker that a certain identifier is legal, and will be present in the program when the DLL is loaded. So the linker uses the information from the import library to build the
lookup table for using identifiers that are not included in the DLL. When an application or a DLL is linked, an import library may be generated, which will need to be used for all future DLLs that depend on the symbols in the application or DLL.

Suppose you are building two dynamic-load modules, B and C, which should share another block of code A. On Unix, you would not pass A.a to the linker for B.so and C.so; that would cause it to be included twice, so that B and C would each have their own copy. In Windows, building A.dll will also build A.lib. You do pass A.lib to the linker for B and C. A.lib does not contain code; it just contains information which will be used at runtime to access A’s code.

In Windows, using an import library is sort of like using import spam; it gives you access to spam’s names, but does not create a separate copy. On Unix, linking with a library is more like from spam import *; it does create a separate copy.
4.3. Using DLLs in Practice

Windows Python is built in Microsoft Visual C++; using other compilers may or may not work (though Borland seems to). The rest of this section is MSVC++ specific.

When creating DLLs in Windows, you must pass pythonXY.lib to the linker. To build two DLLs, spam and ni (which uses C functions found in spam), you could use these commands:

```
cl /LD /I/python/include spam.c ..;/libs/pythonXY.lib
cl /LD /I/python/include ni.c spam.lib ..;/libs/pythonXY.lib
```

The first command created three files: spam.obj, spam.dll and spam.lib. Spam.dll does not contain any Python functions (such as `PyArg_ParseTuple()`), but it does know how to find the Python code thanks to pythonXY.lib.

The second command created ni.dll (and .obj and .lib), which knows how to find the necessary functions from spam, and also from the Python executable.

Not every identifier is exported to the lookup table. If you want any other modules (including Python) to be able to see your identifiers, you have to say `_declspec(dllexport)`, as in `void _declspec(dllexport) initspam(void)` or `PyObject _declspec(dllexport) *NiGetSpamData(void)`.

Developer Studio will throw in a lot of import libraries that you do not really need, adding about 100K to your executable. To get rid of them, use the Project Settings dialog, Link tab, to specify `ignore default libraries`. Add the correct msvcrxx.lib to the list of libraries.
5. Embedding Python in Another Application

The previous chapters discussed how to extend Python, that is, how to extend the functionality of Python by attaching a library of C functions to it. It is also possible to do it the other way around: enrich your C/C++ application by embedding Python in it. Embedding provides your application with the ability to implement some of the functionality of your application in Python rather than C or C++. This can be used for many purposes; one example would be to allow users to tailor the application to their needs by writing some scripts in Python. You can also use it yourself if some of the functionality can be written in Python more easily.

Embedding Python is similar to extending it, but not quite. The difference is that when you extend Python, the main program of the application is still the Python interpreter, while if you embed Python, the main program may have nothing to do with Python — instead, some parts of the application occasionally call the Python interpreter to run some Python code.

So if you are embedding Python, you are providing your own main program. One of the things this main program has to do is initialize the Python interpreter. At the very least, you have to call the function Py_Initialize(). There are optional calls to pass command line arguments to Python. Then later you can call the interpreter from any part of the application.

There are several different ways to call the interpreter: you can pass a string containing Python statements to PyRun_SimpleString(), or you can pass a stdio file pointer and a file name (for identification in error messages only) to PyRun_SimpleFile(). You can also call the lower-level operations described in the previous chapters to
construct and use Python objects.

A simple demo of embedding Python can be found in the directory Demo/embed/ of the source distribution.

See also:

Python/C API Reference Manual
The details of Python’s C interface are given in this manual. A great deal of necessary information can be found here.
5.1. Very High Level Embedding

The simplest form of embedding Python is the use of the very high level interface. This interface is intended to execute a Python script without needing to interact with the application directly. This can for example be used to perform some operation on a file.

```c
#include <Python.h>

int main(int argc, char *argv[])
{
    Py_SetProgramName(argv[0]); /* optional but recommended */
    Py_Initialize();
    PyRun_SimpleString("from time import time,ctime
"                  "print 'Today is',ctime(time())\n");
    Py_Finalize();
    return 0;
}
```

The `Py_SetProgramName()` function should be called before `Py_Initialize()` to inform the interpreter about paths to Python run-time libraries. Next, the Python interpreter is initialized with `Py_Initialize()`, followed by the execution of a hard-coded Python script that prints the date and time. Afterwards, the `Py_Finalize()` call shuts the interpreter down, followed by the end of the program. In a real program, you may want to get the Python script from another source, perhaps a text-editor routine, a file, or a database. Getting the Python code from a file can better be done by using the `PyRun_SimpleFile()` function, which saves you the trouble of allocating memory space and loading the file contents.
5.2. Beyond Very High Level Embedding: An overview

The high level interface gives you the ability to execute arbitrary pieces of Python code from your application, but exchanging data values is quite cumbersome to say the least. If you want that, you should use lower level calls. At the cost of having to write more C code, you can achieve almost anything.

It should be noted that extending Python and embedding Python is quite the same activity, despite the different intent. Most topics discussed in the previous chapters are still valid. To show this, consider what the extension code from Python to C really does:

1. Convert data values from Python to C,
2. Perform a function call to a C routine using the converted values, and
3. Convert the data values from the call from C to Python.

When embedding Python, the interface code does:

1. Convert data values from C to Python,
2. Perform a function call to a Python interface routine using the converted values, and
3. Convert the data values from the call from Python to C.

As you can see, the data conversion steps are simply swapped to accommodate the different direction of the cross-language transfer. The only difference is the routine that you call between both data conversions. When extending, you call a C routine, when embedding, you call a Python routine.

This chapter will not discuss how to convert data from Python to C and vice versa. Also, proper use of references and dealing with
errors is assumed to be understood. Since these aspects do not differ from extending the interpreter, you can refer to earlier chapters for the required information.
5.3. Pure Embedding

The first program aims to execute a function in a Python script. Like in the section about the very high level interface, the Python interpreter does not directly interact with the application (but that will change in the next section).

The code to run a function defined in a Python script is:

```c
#include <Python.h>

int main(int argc, char *argv[]) {
    PyObject *pName, *pModule, *pDict, *pFunc;
    PyObject *pArgs, *pValue;
    int i;

    if (argc < 3) {
        fprintf(stderr, "Usage: call pythonfile funcname [args]\n        return 1;
    }

    Py_Initialize();
    pName = PyString_FromString(argv[1]);
    /* Error checking of pName left out */

    pModule = PyImport_Import(pName);
    Py_DECREF(pName);

    if (pModule != NULL) {
        pFunc = PyObject_GetAttrString(pModule, argv[2]);
        /* pFunc is a new reference */

        if (pFunc && PyCallable_Check(pFunc)) {
            pArgs = PyTuple_New(argc - 3);
            for (i = 0; i < argc - 3; ++i) {
                pValue = PyInt_FromLong(atoi(argv[i + 3]));
                if (!pValue) {
                    Py_DECREF(pArgs);
                    Py_DECREF(pModule);
                    fprintf(stderr, "Cannot convert argument\n";
```
This code loads a Python script using `argv[1]`, and calls the function named in `argv[2]`. Its integer arguments are the other values of the `argv` array. If you compile and link this program (let's call the finished executable `call`), and use it to execute a Python script, such as:

```python
def multiply(a,b):
    print "Will compute", a, "times", b
```
```python
c = 0
for i in range(0, a):
    c = c + b
return c
```

then the result should be:

```
$ call multiply multiply 3 2
Will compute 3 times 2
Result of call: 6
```

Although the program is quite large for its functionality, most of the code is for data conversion between Python and C, and for error reporting. The interesting part with respect to embedding Python starts with

```c
Py_Initialize();
pName = PyString_FromString(argv[1]);
/* Error checking of pName left out */
pModule = PyImport_Import(pName);
```

After initializing the interpreter, the script is loaded using `PyImport_Import()`. This routine needs a Python string as its argument, which is constructed using the `PyString_FromString()` data conversion routine.

```c
pFunc = PyObject_GetAttrString(pModule, argv[2]);
/* pFunc is a new reference */
if (pFunc && PyCallable_Check(pFunc)) {
    ...
}
Py_XDECREF(pFunc);
```

Once the script is loaded, the name we're looking for is retrieved using `PyObject_GetAttrString()`. If the name exists, and the object returned is callable, you can safely assume that it is a function. The program then proceeds by constructing a tuple of arguments as normal. The call to the Python function is then made with:
pValue = PyObject_CallObject(pFunc, pArgs);

Upon return of the function, `pValue` is either `NULL` or it contains a reference to the return value of the function. Be sure to release the reference after examining the value.
5.4. Extending Embedded Python

Until now, the embedded Python interpreter had no access to functionality from the application itself. The Python API allows this by extending the embedded interpreter. That is, the embedded interpreter gets extended with routines provided by the application. While it sounds complex, it is not so bad. Simply forget for a while that the application starts the Python interpreter. Instead, consider the application to be a set of subroutines, and write some glue code that gives Python access to those routines, just like you would write a normal Python extension. For example:

```c
static int numargs=0;

/* Return the number of arguments of the application command line */
static PyObject*
emb_numargs(PyObject *self, PyObject *args)
{
    if(!PyArg_ParseTuple(args, ":numargs"))
        return NULL;
    return Py_BuildValue("i", numargs);
}

static PyMethodDef EmbMethods[] = {
    {"numargs", emb_numargs, METH_VARARGS,
    "Return the number of arguments received by the process."}
    {NULL, NULL, 0, NULL}
};
```

Insert the above code just above the `main()` function. Also, insert the following two statements directly after `Py_Initialize()`:

```c
numargs = argc;
Py_InitModule("emb", EmbMethods);
```

These two lines initialize the `numargs` variable, and make the `emb.numargs()` function accessible to the embedded Python
interpreter. With these extensions, the Python script can do things like

```python
import emb
print "Number of arguments", emb.numargs()
```

In a real application, the methods will expose an API of the application to Python.
5.5. Embedding Python in C++

It is also possible to embed Python in a C++ program; precisely how this is done will depend on the details of the C++ system used; in general you will need to write the main program in C++, and use the C++ compiler to compile and link your program. There is no need to recompile Python itself using C++. 
5.6. Compiling and Linking under Unix-like systems

It is not necessarily trivial to find the right flags to pass to your compiler (and linker) in order to embed the Python interpreter into your application, particularly because Python needs to load library modules implemented as C dynamic extensions (.so files) linked against it.

To find out the required compiler and linker flags, you can execute the `pythonX.Y-config` script which is generated as part of the installation process (a `python-config` script may also be available). This script has several options, of which the following will be directly useful to you:

- `pythonX.Y-config --cflags` will give you the recommended flags when compiling:

  ```bash
  $ /opt/bin/python2.7-config --cflags
  -I/opt/include/python2.7 -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes
  ```

- `pythonX.Y-config --ldflags` will give you the recommended flags when linking:

  ```bash
  $ /opt/bin/python2.7-config --ldflags
  -L/opt/lib/python2.7/config -lpthread -ldl -lutil -lm -lpython2.7
  ```

**Note:** To avoid confusion between several Python installations (and especially between the system Python and your own compiled Python), it is recommended that you use the absolute path to `pythonX.Y-config`, as in the above example.
If this procedure doesn’t work for you (it is not guaranteed to work for all Unix-like platforms; however, we welcome bug reports) you will have to read your system’s documentation about dynamic linking and/or examine Python’s Makefile (use \texttt{sysconfig.get_makefile_filename()} to find its location) and compilation options. In this case, the \texttt{sysconfig} module is a useful tool to programmatically extract the configuration values that you will want to combine together. For example:

\begin{verbatim}
>>> import sysconfig
>>> sysconfig.get_config_var('LIBS')
'-lpthread -ldl -lutil'
>>> sysconfig.get_config_var('LINKFORSHARED')
'-Xlinker -export-dynamic'
\end{verbatim}
The Application Programmer’s Interface to Python gives C and C++ programmers access to the Python interpreter at a variety of levels. The API is equally usable from C++, but for brevity it is generally referred to as the Python/C API. There are two fundamentally different reasons for using the Python/C API. The first reason is to write extension modules for specific purposes; these are C modules that extend the Python interpreter. This is probably the most common use. The second reason is to use Python as a component in a larger application; this technique is generally referred to as embedding Python in an application.

Writing an extension module is a relatively well-understood process, where a “cookbook” approach works well. There are several tools that automate the process to some extent. While people have embedded Python in other applications since its early existence, the process of embedding Python is less straightforward than writing an extension.

Many API functions are useful independent of whether you’re embedding or extending Python; moreover, most applications that embed Python will need to provide a custom extension as well, so it’s probably a good idea to become familiar with writing an extension before attempting to embed Python in a real application.
Include Files

All function, type and macro definitions needed to use the Python/C API are included in your code by the following line:

```
#include "Python.h"
```

This implies inclusion of the following standard headers: `<stdio.h>`, `<string.h>`, `<errno.h>`, `<limits.h>`, `<assert.h>` and `<stdlib.h>` (if available).

**Note:** Since Python may define some pre-processor definitions which affect the standard headers on some systems, you *must* include `Python.h` before any standard headers are included.

All user visible names defined by `Python.h` (except those defined by the included standard headers) have one of the prefixes `Py` or `_Py`. Names beginning with `_Py` are for internal use by the Python implementation and should not be used by extension writers. Structure member names do not have a reserved prefix.

**Important:** user code should never define names that begin with `Py` or `_Py`. This confuses the reader, and jeopardizes the portability of the user code to future Python versions, which may define additional names beginning with one of these prefixes.

The header files are typically installed with Python. On Unix, these are located in the directories `prefix/include/pythonversion/` and `exec_prefix/include/pythonversion/`, where `prefix` and `exec_prefix` are defined by the corresponding parameters to Python’s `configure` script and `version` is `sys.version[:3]`. On Windows, the headers are installed in `prefix/include`, where `prefix`
is the installation directory specified to the installer.

To include the headers, place both directories (if different) on your compiler’s search path for includes. Do *not* place the parent directories on the search path and then use `#include <pythonX.Y/Python.h>`; this will break on multi-platform builds since the platform independent headers under `prefix` include the platform specific headers from `exec_prefix`.

C++ users should note that though the API is defined entirely using C, the header files do properly declare the entry points to be `extern "C"`, so there is no need to do anything special to use the API from C++.
Objects, Types and Reference Counts

Most Python/C API functions have one or more arguments as well as a return value of type `PyObject*`. This type is a pointer to an opaque data type representing an arbitrary Python object. Since all Python object types are treated the same way by the Python language in most situations (e.g., assignments, scope rules, and argument passing), it is only fitting that they should be represented by a single C type. Almost all Python objects live on the heap: you never declare an automatic or static variable of type `PyObject`, only pointer variables of type `PyObject*` can be declared. The sole exception are the type objects; since these must never be deallocated, they are typically static `PyTypeObject` objects.

All Python objects (even Python integers) have a type and a reference count. An object’s type determines what kind of object it is (e.g., an integer, a list, or a user-defined function; there are many more as explained in The standard type hierarchy). For each of the well-known types there is a macro to check whether an object is of that type; for instance, `PyList_Check(a)` is true if (and only if) the object pointed to by a is a Python list.

Reference Counts

The reference count is important because today’s computers have a finite (and often severely limited) memory size; it counts how many different places there are that have a reference to an object. Such a place could be another object, or a global (or static) C variable, or a local variable in some C function. When an object’s reference count becomes zero, the object is deallocated. If it contains references to other objects, their reference count is decremented. Those other objects may be deallocated in turn, if this decrement makes their
reference count become zero, and so on. (There’s an obvious problem with objects that reference each other here; for now, the solution is “don’t do that.”)

Reference counts are always manipulated explicitly. The normal way is to use the macro \texttt{Py_INCREF()} to increment an object’s reference count by one, and \texttt{Py_DECREF()} to decrement it by one. The \texttt{Py_DECREF()} macro is considerably more complex than the incref one, since it must check whether the reference count becomes zero and then cause the object’s deallocator to be called. The deallocator is a function pointer contained in the object’s type structure. The type-specific deallocator takes care of decrementing the reference counts for other objects contained in the object if this is a compound object type, such as a list, as well as performing any additional finalization that’s needed. There’s no chance that the reference count can overflow; at least as many bits are used to hold the reference count as there are distinct memory locations in virtual memory (assuming $\text{sizeof}(\text{Py_ssize_t}) \geq \text{sizeof(void*)}$). Thus, the reference count increment is a simple operation.

It is not necessary to increment an object’s reference count for every local variable that contains a pointer to an object. In theory, the object’s reference count goes up by one when the variable is made to point to it and it goes down by one when the variable goes out of scope. However, these two cancel each other out, so at the end the reference count hasn’t changed. The only real reason to use the reference count is to prevent the object from being deallocated as long as our variable is pointing to it. If we know that there is at least one other reference to the object that lives at least as long as our variable, there is no need to increment the reference count temporarily. An important situation where this arises is in objects that are passed as arguments to C functions in an extension module that are called from Python; the call mechanism guarantees to hold a reference to every argument for the duration of the call.
However, a common pitfall is to extract an object from a list and hold on to it for a while without incrementing its reference count. Some other operation might conceivably remove the object from the list, decrementing its reference count and possible deallocating it. The real danger is that innocent-looking operations may invoke arbitrary Python code which could do this; there is a code path which allows control to flow back to the user from a \texttt{Py\_DECREF()}, so almost any operation is potentially dangerous.

A safe approach is to always use the generic operations (functions whose name begins with \texttt{PyObject\_}, \texttt{PyNumber\_}, \texttt{PySequence\_} or \texttt{PyMapping\_}). These operations always increment the reference count of the object they return. This leaves the caller with the responsibility to call \texttt{Py\_DECREF()} when they are done with the result; this soon becomes second nature.

### Reference Count Details

The reference count behavior of functions in the Python/C API is best explained in terms of \textit{ownership of references}. Ownership pertains to references, never to objects (objects are not owned: they are always shared). “Owning a reference” means being responsible for calling \texttt{Py\_DECREF} on it when the reference is no longer needed. Ownership can also be transferred, meaning that the code that receives ownership of the reference then becomes responsible for eventually decref’ing it by calling \texttt{Py\_DECREF()} or \texttt{Py\_XDECREF()} when it’s no longer needed—or passing on this responsibility (usually to its caller). When a function passes ownership of a reference on to its caller, the caller is said to receive a \textit{new} reference. When no ownership is transferred, the caller is said to \textit{borrow} the reference. Nothing needs to be done for a borrowed reference.

Conversely, when a calling function passes in a reference to an
object, there are two possibilities: the function steals a reference to the object, or it does not. Stealing a reference means that when you pass a reference to a function, that function assumes that it now owns that reference, and you are not responsible for it any longer.

Few functions steal references; the two notable exceptions are `PyList_SetItem()` and `PyTuple_SetItem()`, which steal a reference to the item (but not to the tuple or list into which the item is put!). These functions were designed to steal a reference because of a common idiom for populating a tuple or list with newly created objects; for example, the code to create the tuple `(1, 2, "three")` could look like this (forgetting about error handling for the moment; a better way to code this is shown below):

```python
PyObject *t;
t = PyTuple_New(3);
PyTuple_SetItem(t, 0, PyInt_FromLong(1L));
PyTuple_SetItem(t, 1, PyInt_FromLong(2L));
PyTuple_SetItem(t, 2, PyString_FromString("three"));
```

Here, `PyInt_FromLong()` returns a new reference which is immediately stolen by `PyTuple_SetItem()`. When you want to keep using an object although the reference to it will be stolen, use `Py_INCREF()` to grab another reference before calling the reference-stealing function.

Incidentally, `PyTuple_SetItem()` is the only way to set tuple items; `PySequence_SetItem()` and `PyObject_SetItem()` refuse to do this since tuples are an immutable data type. You should only use `PyTuple_SetItem()` for tuples that you are creating yourself.

Equivalent code for populating a list can be written using `PyList_New()` and `PyList_SetItem()`.

However, in practice, you will rarely use these ways of creating and
populating a tuple or list. There's a generic function, `Py_BuildValue()`, that can create most common objects from C values, directed by a *format string*. For example, the above two blocks of code could be replaced by the following (which also takes care of the error checking):

```c
PyObject *tuple, *list;
tuple = Py_BuildValue("(iis)", 1, 2, "three");
list = Py_BuildValue("[iis]", 1, 2, "three");
```

It is much more common to use `PyObject_SetItem()` and friends with items whose references you are only borrowing, like arguments that were passed in to the function you are writing. In that case, their behaviour regarding reference counts is much saner, since you don’t have to increment a reference count so you can give a reference away (“have it be stolen”). For example, this function sets all items of a list (actually, any mutable sequence) to a given item:

```c
int
set_all(PyObject *target, PyObject *item)
{
    int i, n;

    n = PyObject_Length(target);
    if (n < 0)
        return -1;
    for (i = 0; i < n; i++) {
        PyObject *index = PyInt_FromLong(i);
        if (!index)
            return -1;
        if (PyObject_SetItem(target, index, item) < 0) {
            Py_DECREF(index);
            return -1;
        }
    }
    Py_DECREF(index);
    return 0;
}
```
The situation is slightly different for function return values. While passing a reference to most functions does not change your ownership responsibilities for that reference, many functions that return a reference to an object give you ownership of the reference. The reason is simple: in many cases, the returned object is created on the fly, and the reference you get is the only reference to the object. Therefore, the generic functions that return object references, like `PyObject_GetItem()` and `PySequence_GetItem()`, always return a new reference (the caller becomes the owner of the reference).

It is important to realize that whether you own a reference returned by a function depends on which function you call only — the plumage (the type of the object passed as an argument to the function) doesn’t enter into it! Thus, if you extract an item from a list using `PyList_GetItem()`, you don’t own the reference — but if you obtain the same item from the same list using `PySequence_GetItem()` (which happens to take exactly the same arguments), you do own a reference to the returned object.

Here is an example of how you could write a function that computes the sum of the items in a list of integers; once using `PyList_GetItem()`, and once using `PySequence_GetItem()`.

```c
long sum_list(PyObject *list)
{
    int i, n;
    long total = 0;
    PyObject *item;

    n = PyList_Size(list);
    if (n < 0)
        return -1; /* Not a list */
    for (i = 0; i < n; i++) {
        item = PyList_GetItem(list, i); /* Can't fail */
        if (!PyInt_Check(item)) continue; /* Skip non-integers */
        total += PyInt_AsLong(item);
    }
}
```
Types

There are few other data types that play a significant role in the Python/C API; most are simple C types such as `int`, `long`, `double` and `char*`. A few structure types are used to describe static tables used to list the functions exported by a module or the data attributes of a new object type, and another is used to describe the value of a complex number. These will be discussed together with the functions that use them.
The Python programmer only needs to deal with exceptions if specific error handling is required; unhandled exceptions are automatically propagated to the caller, then to the caller’s caller, and so on, until they reach the top-level interpreter, where they are reported to the user accompanied by a stack traceback.

For C programmers, however, error checking always has to be explicit. All functions in the Python/C API can raise exceptions, unless an explicit claim is made otherwise in a function’s documentation. In general, when a function encounters an error, it sets an exception, discards any object references that it owns, and returns an error indicator. If not documented otherwise, this indicator is either `NULL` or `-1`, depending on the function’s return type. A few functions return a Boolean true/false result, with false indicating an error. Very few functions return no explicit error indicator or have an ambiguous return value, and require explicit testing for errors with `PyErr_Occurred()`. These exceptions are always explicitly documented.

Exception state is maintained in per-thread storage (this is equivalent to using global storage in an unthreaded application). A thread can be in one of two states: an exception has occurred, or not. The function `PyErr_Occurred()` can be used to check for this: it returns a borrowed reference to the exception type object when an exception has occurred, and `NULL` otherwise. There are a number of functions to set the exception state: `PyErr_SetString()` is the most common (though not the most general) function to set the exception state, and `PyErr_Clear()` clears the exception state.

The full exception state consists of three objects (all of which can be `NULL`): the exception type, the corresponding exception value, and
the traceback. These have the same meanings as the Python objects `sys.exc_type`, `sys.exc_value`, and `sys.exc_traceback`; however, they are not the same: the Python objects represent the last exception being handled by a Python `try ... except` statement, while the C level exception state only exists while an exception is being passed on between C functions until it reaches the Python bytecode interpreter’s main loop, which takes care of transferring it to `sys.exc_type` and friends.

Note that starting with Python 1.5, the preferred, thread-safe way to access the exception state from Python code is to call the function `sys.exc_info()`, which returns the per-thread exception state for Python code. Also, the semantics of both ways to access the exception state have changed so that a function which catches an exception will save and restore its thread’s exception state so as to preserve the exception state of its caller. This prevents common bugs in exception handling code caused by an innocent-looking function overwriting the exception being handled; it also reduces the often unwanted lifetime extension for objects that are referenced by the stack frames in the traceback.

As a general principle, a function that calls another function to perform some task should check whether the called function raised an exception, and if so, pass the exception state on to its caller. It should discard any object references that it owns, and return an error indicator, but it should not set another exception — that would overwrite the exception that was just raised, and lose important information about the exact cause of the error.

A simple example of detecting exceptions and passing them on is shown in the `sum_sequence()` example above. It so happens that this example doesn’t need to clean up any owned references when it detects an error. The following example function shows some error cleanup. First, to remind you why you like Python, we show the
equivalent Python code:

def incr_item(dict, key):
    try:
        item = dict[key]
    except KeyError:
        item = 0
    dict[key] = item + 1

Here is the corresponding C code, in all its glory:

```c
int
incr_item(PyObject *dict, PyObject *key)
{
    /* Objects all initialized to NULL for Py_XDECREF */
    PyObject *item = NULL, *const_one = NULL, *incremented_item
    int rv = -1; /* Return value initialized to -1 (failure) */

    item = PyObject_GetItem(dict, key);
    if (item == NULL) {
        /* Handle KeyError only: */
        if (!PyErr_ExceptionMatches(PyExc_KeyError))
            goto error;

        /* Clear the error and use zero: */
        PyErr_Clear();
        item = PyInt_FromLong(0L);
        if (item == NULL)
            goto error;
    }
    const_one = PyInt_FromLong(1L);
    if (const_one == NULL)
        goto error;

    incremented_item = PyNumber_Add(item, const_one);
    if (incremented_item == NULL)
        goto error;

    if (PyObject_SetItem(dict, key, incremented_item) < 0)
        goto error;
    rv = 0; /* Success */
    /* Continue with cleanup code */

error:
    /* Cleanup code, shared by success and failure path */
```
This example represents an endorsed use of the `goto` statement in C! It illustrates the use of `PyErr_ExceptionMatches()` and `PyErr_Clear()` to handle specific exceptions, and the use of `Py_XDECRREF()` to dispose of owned references that may be `NULL` (note the 'x' in the name; `Py_DECREF()` would crash when confronted with a `NULL` reference). It is important that the variables used to hold owned references are initialized to `NULL` for this to work; likewise, the proposed return value is initialized to `-1` (failure) and only set to success after the final call made is successful.
The one important task that only embedders (as opposed to extension writers) of the Python interpreter have to worry about is the initialization, and possibly the finalization, of the Python interpreter. Most functionality of the interpreter can only be used after the interpreter has been initialized.

The basic initialization function is `Py_Initialize()`. This initializes the table of loaded modules, and creates the fundamental modules `__builtin__`, `__main__`, `sys`, and `exceptions`. It also initializes the module search path (`sys.path`).

`Py_Initialize()` does not set the “script argument list” (`sys.argv`). If this variable is needed by Python code that will be executed later, it must be set explicitly with a call to `PySys_SetArgvEx(argc, argv, updatepath)` after the call to `Py_Initialize()`.

On most systems (in particular, on Unix and Windows, although the details are slightly different), `Py_Initialize()` calculates the module search path based upon its best guess for the location of the standard Python interpreter executable, assuming that the Python library is found in a fixed location relative to the Python interpreter executable. In particular, it looks for a directory named `lib/pythonX.Y` relative to the parent directory where the executable named `python` is found on the shell command search path (the environment variable `PATH`).

For instance, if the Python executable is found in `/usr/local/bin/python`, it will assume that the libraries are in `/usr/local/lib/pythonX.Y`. (In fact, this particular path is also the “fallback” location, used when no executable file named `python` is
found along **PATH**.) The user can override this behavior by setting the environment variable **PYTHONHOME**, or insert additional directories in front of the standard path by setting **PYTHONPATH**.

The embedding application can steer the search by calling `Py_SetProgramName(file)` before calling `Py_Initialize()`. Note that **PYTHONHOME** still overrides this and **PYTHONPATH** is still inserted in front of the standard path. An application that requires total control has to provide its own implementation of `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, and `Py_GetProgramFullPath()` (all defined in Modules/getpath.c).

Sometimes, it is desirable to “uninitialize” Python. For instance, the application may want to start over (make another call to `Py_Initialize()`) or the application is simply done with its use of Python and wants to free memory allocated by Python. This can be accomplished by calling `Py_Finalize()`. The function `Py_IsInitialized()` returns true if Python is currently in the initialized state. More information about these functions is given in a later chapter. Notice that `Py_Finalize()` does not free all memory allocated by the Python interpreter, e.g. memory allocated by extension modules currently cannot be released.
Debugging Builds

Python can be built with several macros to enable extra checks of the interpreter and extension modules. These checks tend to add a large amount of overhead to the runtime so they are not enabled by default.

A full list of the various types of debugging builds is in the file Misc/SpecialBuilds.txt in the Python source distribution. Builds are available that support tracing of reference counts, debugging the memory allocator, or low-level profiling of the main interpreter loop. Only the most frequently-used builds will be described in the remainder of this section.

Compiling the interpreter with the Py_DEBUG macro defined produces what is generally meant by “a debug build” of Python. Py_DEBUG is enabled in the Unix build by adding --with-pydebug to the ./configure command. It is also implied by the presence of the not-Python-specific _DEBUG macro. When Py_DEBUG is enabled in the Unix build, compiler optimization is disabled.

In addition to the reference count debugging described below, the following extra checks are performed:

- Extra checks are added to the object allocator.
- Extra checks are added to the parser and compiler.
- Downcasts from wide types to narrow types are checked for loss of information.
- A number of assertions are added to the dictionary and set implementations. In addition, the set object acquires a test_c_api() method.
- Sanity checks of the input arguments are added to frame creation.
The storage for long ints is initialized with a known invalid pattern to catch reference to uninitialized digits.
Low-level tracing and extra exception checking are added to the runtime virtual machine.
Extra checks are added to the memory arena implementation.
Extra debugging is added to the thread module.

There may be additional checks not mentioned here.

Defining `Py_TRACE_REFS` enables reference tracing. When defined, a circular doubly linked list of active objects is maintained by adding two extra fields to every `PyObject`. Total allocations are tracked as well. Upon exit, all existing references are printed. (In interactive mode this happens after every statement run by the interpreter.) Implied by `Py_DEBUG`.

Please refer to `Misc/SpecialBuilds.txt` in the Python source distribution for more detailed information.
The Very High Level Layer

The functions in this chapter will let you execute Python source code given in a file or a buffer, but they will not let you interact in a more detailed way with the interpreter.

Several of these functions accept a start symbol from the grammar as a parameter. The available start symbols are `Py_eval_input`, `Py_file_input`, and `Py_single_input`. These are described following the functions which accept them as parameters.

Note also that several of these functions take `FILE*` parameters. One particular issue which needs to be handled carefully is that the `FILE` structure for different C libraries can be different and incompatible. Under Windows (at least), it is possible for dynamically linked extensions to actually use different libraries, so care should be taken that `FILE*` parameters are only passed to these functions if it is certain that they were created by the same library that the Python runtime is using.

```c
int Py_Main(int argc, char **argv)
```

The main program for the standard interpreter. This is made available for programs which embed Python. The `argc` and `argv` parameters should be prepared exactly as those which are passed to a C program's `main()` function. It is important to note that the argument list may be modified (but the contents of the strings pointed to by the argument list are not). The return value will be 0 if the interpreter exits normally (ie, without an exception), 1 if the interpreter exits due to an exception, or 2 if the parameter list does not represent a valid Python command line.

Note that if an otherwise unhandled `SystemExit` is raised, this function will not return 1, but exit the process, as long as
Py_InspectFlag is not set.

int PyRun<AnyFile(FILE *fp, const char *filename)
    This is a simplified interface to PyRun<AnyFileExFlags() below, leaving closeit set to 0 and flags set to NULL.

int PyRun<AnyFileFlags(FILE *fp, const char *filename, PyCompilerFlags *flags)
    This is a simplified interface to PyRun<AnyFileExFlags() below, leaving the closeit argument set to 0.

int PyRun<AnyFileEx(FILE *fp, const char *filename, int closeit)
    This is a simplified interface to PyRun<AnyFileExFlags() below, leaving the flags argument set to NULL.

int PyRun<AnyFileExFlags(FILE *fp, const char *filename, int closeit, PyCompilerFlags *flags)
    If fp refers to a file associated with an interactive device (console or terminal input or Unix pseudo-terminal), return the value of PyRun_InteractiveLoop(), otherwise return the result of PyRun_SimpleFile(). If filename is NULL, this function uses "???", as the filename.

int PyRun_SimpleString(const char *command)
    This is a simplified interface to PyRun_SimpleStringFlags() below, leaving the PyCompilerFlags* argument set to NULL.

int PyRun_SimpleStringFlags(const char *command, PyCompilerFlags *flags)
    Executes the Python source code from command in the __main__ module according to the flags argument. If __main__ does not already exist, it is created. Returns 0 on success or -1 if an exception was raised. If there was an error, there is no way to get the exception information. For the meaning of flags, see below.
Note that if an otherwise unhandled `SystemExit` is raised, this function will not return `-1`, but exit the process, as long as `Py_InspectFlag` is not set.

```c
int PyRun_SimpleFile(FILE *fp, const char *filename)
    This is a simplified interface to `PyRun_SimpleFileExFlags()` below, leaving `closeit` set to 0 and `flags` set to `NULL`.
```

```c
int PyRun_SimpleFileFlags(FILE *fp, const char *filename, PyCompilerFlags *flags)
    This is a simplified interface to `PyRun_SimpleFileExFlags()` below, leaving `closeit` set to 0.
```

```c
int PyRun_SimpleFileEx(FILE *fp, const char *filename, int closeit)
    This is a simplified interface to `PyRun_SimpleFileExFlags()` below, leaving `flags` set to `NULL`.
```

```c
int PyRun_SimpleFileExFlags(FILE *fp, const char *filename, int closeit, PyCompilerFlags *flags)
    Similar to `PyRun_SimpleStringFlags()`, but the Python source code is read from `fp` instead of an in-memory string. `filename` should be the name of the file. If `closeit` is true, the file is closed before `PyRun_SimpleFileExFlags` returns.
```

```c
int PyRun_InteractiveOne(FILE *fp, const char *filename)
    This is a simplified interface to `PyRun_InteractiveOneFlags()` below, leaving `flags` set to `NULL`.
```

```c
int PyRun_InteractiveOneFlags(FILE *fp, const char *filename, PyCompilerFlags *flags)
    Read and execute a single statement from a file associated with an interactive device according to the `flags` argument. The user will be prompted using `sys.ps1` and `sys.ps2`. Returns 0 when the input was executed successfully, -1 if there was an exception, or
an error code from the errcode.h include file distributed as part of Python if there was a parse error. (Note that errcode.h is not included by Python.h, so must be included specifically if needed.)

int PyRun_InteractiveLoop(FILE *fp, const char *filename)
This is a simplified interface to PyRun_InteractiveLoopFlags() below, leaving flags set to NULL.

int PyRun_InteractiveLoopFlags(FILE *fp, const char *filename, PyCompilerFlags *flags)
Read and execute statements from a file associated with an interactive device until EOF is reached. The user will be prompted using sys.ps1 and sys.ps2. Returns 0 at EOF.

struct _node* PyParser_SimpleParseString(const char *str, int start)
This is a simplified interface to PyParser_SimpleParseStringFlagsFilename() below, leaving filename set to NULL and flags set to 0.

struct _node* PyParser_SimpleParseStringFlags(const char *str, int start, int flags)
This is a simplified interface to PyParser_SimpleParseStringFlagsFilename() below, leaving filename set to NULL.

struct _node* PyParser_SimpleParseStringFlagsFilename(const char *str, const char *filename, int start, int flags)
Parse Python source code from str using the start token start according to the flags argument. The result can be used to create a code object which can be evaluated efficiently. This is useful if a code fragment must be evaluated many times.

struct _node* PyParser_SimpleParseFile(FILE *fp, const char *filename, int start)
This is a simplified interface to `PyParser_SimpleParseFileFlags()` below, leaving `flags` set to 0.

```c
struct_node* PyParser_SimpleParseFileFlags(FILE *fp, const char *filename, int start, int flags)
```

Similar to `PyParser_SimpleParseStringFlagsFilename()`, but the Python source code is read from `fp` instead of an in-memory string.

```c
PyObject* PyRun_String(const char *str, int start, PyObject *globals, PyObject *locals)
```

*Return value: New reference.*

This is a simplified interface to `PyRun_StringFlags()` below, leaving `flags` set to NULL.

```c
PyObject* PyRun_StringFlags(const char *str, int start, PyObject *globals, PyObject *locals, PyCompilerFlags *flags)
```

*Return value: New reference.*

Execute Python source code from `str` in the context specified by the dictionaries `globals` and `locals` with the compiler flags specified by `flags`. The parameter `start` specifies the start token that should be used to parse the source code.

Returns the result of executing the code as a Python object, or `NULL` if an exception was raised.

```c
PyObject* PyRun_File(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals)
```

*Return value: New reference.*

This is a simplified interface to `PyRun_FileExFlags()` below, leaving `closeit` set to 0 and `flags` set to NULL.

```c
PyObject* PyRun_FileEx(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, int closeit)
```

*Return value: New reference.*
This is a simplified interface to `PyRun_FileExFlags()` below, leaving `flags` set to `NULL`.

```
PyObject* PyRun_FileFlags(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, PyCompilerFlags *flags)
   This is a simplified interface to `PyRun_FileExFlags()` below, leaving `closeit` set to 0.
```

```
PyObject* PyRun_FileExFlags(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, int closeit, PyCompilerFlags *flags)
   Similar to `PyRun_StringFlags()`, but the Python source code is read from `fp` instead of an in-memory string. `filename` should be the name of the file. If `closeit` is true, the file is closed before `PyRun_FileExFlags()` returns.
```

```
PyObject* Py_CompileString(const char *str, const char *filename, int start)
   This is a simplified interface to `Py_CompileStringFlags()` below, leaving `flags` set to `NULL`.
```

```
PyObject* Py_CompileStringFlags(const char *str, const char *filename, int start, PyCompilerFlags *flags)
   Parse and compile the Python source code in `str`, returning the resulting code object. The start token is given by `start`; this can be used to constrain the code which can be compiled and should be `Py_eval_input`, `Py_file_input`, or `Py_single_input`. The filename specified by `filename` is used to construct the code object and may appear in traceback or `SyntaxError` exception messages. This returns `NULL` if the code cannot be parsed or compiled.
```
PyObject* PyEval_EvalCode(PyCodeObject *co, PyObject *globals, PyObject *locals)

This is a simplified interface to PyEval_EvalCodeEx(), with just the code object, and the dictionaries of global and local variables. The other arguments are set to NULL.

PyObject* PyEval_EvalCodeEx(PyCodeObject *co, PyObject *globals, PyObject *locals, PyObject **args, int argcount, PyObject **kws, int kwcount, PyObject **defs, int defcount, PyObject *closure)

Evaluate a precompiled code object, given a particular environment for its evaluation. This environment consists of dictionaries of global and local variables, arrays of arguments, keywords and defaults, and a closure tuple of cells.

PyObject* PyEval_EvalFrame(PyFrameObject *f)

Evaluate an execution frame. This is a simplified interface to PyEval_EvalFrameEx, for backward compatibility.

PyObject* PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)

This is the main, unvarnished function of Python interpretation. It is literally 2000 lines long. The code object associated with the execution frame f is executed, interpreting bytecode and executing calls as needed. The additional throwflag parameter can mostly be ignored - if true, then it causes an exception to immediately be thrown; this is used for the throw() methods of generator objects.

int PyEval_MergeCompilerFlags(PyCompilerFlags *cf)

This function changes the flags of the current evaluation frame, and returns true on success, false on failure.

int Py_eval_input

The start symbol from the Python grammar for isolated
expressions; for use with Py_CompileString().

int Py_file_input
The start symbol from the Python grammar for sequences of statements as read from a file or other source; for use with Py_CompileString(). This is the symbol to use when compiling arbitrarily long Python source code.

int Py_single_input
The start symbol from the Python grammar for a single statement; for use with Py_CompileString(). This is the symbol used for the interactive interpreter loop.

struct PyCompilerFlags
This is the structure used to hold compiler flags. In cases where code is only being compiled, it is passed as int flags, and in cases where code is being executed, it is passed as PyCompilerFlags *flags. In this case, from __future__ import can modify flags.

Whenever PyCompilerFlags *flags is NULL, cf_flags is treated as equal to 0, and any modification due to from __future__ import is discarded.

\[
\textbf{struct} \hspace{1em} \text{PyCompilerFlags} \{ \\
\hspace{2em} \text{int} \hspace{1em} \text{cf_flags}; \\
\}
\]

int CO_FUTURE_DIVISION
This bit can be set in flags to cause division operator / to be interpreted as “true division” according to PEP 238.
Reference Counting

The macros in this section are used for managing reference counts of Python objects.

```c
void Py_INCREF(PyObject *o)
```
Increment the reference count for object o. The object must not be `NULL`; if you aren't sure that it isn't `NULL`, use `Py_XINCREF()`.

```c
void Py_XINCREF(PyObject *o)
```
Increment the reference count for object o. The object may be `NULL`, in which case the macro has no effect.

```c
void Py_DECREF(PyObject *o)
```
Decrement the reference count for object o. The object must not be `NULL`; if you aren't sure that it isn't `NULL`, use `Py_XDECREF()`. If the reference count reaches zero, the object's type's deallocation function (which must not be `NULL`) is invoked.

**Warning:** The deallocation function can cause arbitrary Python code to be invoked (e.g. when a class instance with a `__del__()` method is deallocated). While exceptions in such code are not propagated, the executed code has free access to all Python global variables. This means that any object that is reachable from a global variable should be in a consistent state before `Py_DECREF()` is invoked. For example, code to delete an object from a list should copy a reference to the deleted object in a temporary variable, update the list data structure, and then call `Py_DECREF()` for the temporary variable.

```c
void Py_XDECREF(PyObject *o)
```
Decrement the reference count for object o. The object may be `NULL`, in which case the macro has no effect; otherwise the
effect is the same as for `Py_DECREF()`, and the same warning applies.

```c
void Py_CLEAR(PyObject *o)
```

Decrement the reference count for object `o`. The object may be `NULL`, in which case the macro has no effect; otherwise the effect is the same as for `Py_DECREF()`, except that the argument is also set to `NULL`. The warning for `Py_DECREF()` does not apply with respect to the object passed because the macro carefully uses a temporary variable and sets the argument to `NULL` before decrementing its reference count.

It is a good idea to use this macro whenever decrementing the value of a variable that might be traversed during garbage collection.

*New in version 2.4.*

The following functions are for runtime dynamic embedding of Python: `Py_IncRef(PyObject *o)`, `Py_DecRef(PyObject *o)`. They are simply exported function versions of `Py_XINCREF()` and `Py_XDECREF()`, respectively.

The following functions or macros are only for use within the interpreter core: `_Py_Dealloc()`, `_Py_ForgetReference()`, `_Py_NewReference()`, as well as the global variable `_Py_RefTotal`.
Exception Handling

The functions described in this chapter will let you handle and raise Python exceptions. It is important to understand some of the basics of Python exception handling. It works somewhat like the Unix \texttt{errno} variable: there is a global indicator (per thread) of the last error that occurred. Most functions don't clear this on success, but will set it to indicate the cause of the error on failure. Most functions also return an error indicator, usually \texttt{NULL} if they are supposed to return a pointer, or \texttt{-1} if they return an integer (exception: the \texttt{PyArg_*()} functions return \texttt{1} for success and \texttt{0} for failure).

When a function must fail because some function it called failed, it generally doesn't set the error indicator; the function it called already set it. It is responsible for either handling the error and clearing the exception or returning after cleaning up any resources it holds (such as object references or memory allocations); it should \textit{not} continue normally if it is not prepared to handle the error. If returning due to an error, it is important to indicate to the caller that an error has been set. If the error is not handled or carefully propagated, additional calls into the Python/C API may not behave as intended and may fail in mysterious ways.

The error indicator consists of three Python objects corresponding to the Python variables \texttt{sys.exc_type, sys.exc_value} and \texttt{sys.exc_traceback}. API functions exist to interact with the error indicator in various ways. There is a separate error indicator for each thread.

\texttt{void PyErr_PrintEx(int set_sys_last_vars)}

Print a standard traceback to \texttt{sys.stderr} and clear the error indicator. Call this function only when the error indicator is set. (Otherwise it will cause a fatal error!)
If `set_sys_last_vars` is nonzero, the variables `sys.last_type`, `sys.last_value` and `sys.last_traceback` will be set to the type, value and traceback of the printed exception, respectively.

```c
void PyErr_Print()
    Alias for PyErr_PrintEx(1).
```

```c
PyObject* PyErr_Occurred()
    Return value: Borrowed reference.
    Test whether the error indicator is set. If set, return the exception type (the first argument to the last call to one of the PyErr_Set*() functions or to PyErr_Restore()). If not set, return NULL. You do not own a reference to the return value, so you do not need to Py_DECREF() it.

    **Note:** Do not compare the return value to a specific exception; use PyErr_ExceptionMatches() instead, shown below. (The comparison could easily fail since the exception may be an instance instead of a class, in the case of a class exception, or it may be a subclass of the expected exception.)
```

```c
int PyErr_ExceptionMatches(PyObject *exc)
    Equivalent to PyErr_GivenExceptionMatches(PyErr_Occurred(), exc). This should only be called when an exception is actually set; a memory access violation will occur if no exception has been raised.
```

```c
int PyErr_GivenExceptionMatches(PyObject *given, PyObject *exc)
    Return true if the `given` exception matches the exception in `exc`. If `exc` is a class object, this also returns true when `given` is an instance of a subclass. If `exc` is a tuple, all exceptions in the tuple (and recursively in subtuples) are searched for a match.
```
void PyErr_NormalizeException(PyObject**exc, PyObject**val, PyObject**tb)
    Under certain circumstances, the values returned by PyErr_Fetch() below can be “unnormalized”, meaning that *exc is a class object but *val is not an instance of the same class. This function can be used to instantiate the class in that case. If the values are already normalized, nothing happens. The delayed normalization is implemented to improve performance.

void PyErr_Clear()
    Clear the error indicator. If the error indicator is not set, there is no effect.

void PyErr_Fetch(PyObject **ptype, PyObject **pvalue, PyObject **ptraceback)
    Retrieve the error indicator into three variables whose addresses are passed. If the error indicator is not set, set all three variables to NULL. If it is set, it will be cleared and you own a reference to each object retrieved. The value and traceback object may be NULL even when the type object is not.

Note: This function is normally only used by code that needs to handle exceptions or by code that needs to save and restore the error indicator temporarily.

void PyErr_Restore(PyObject *type, PyObject *value, PyObject *traceback)
    Set the error indicator from the three objects. If the error indicator is already set, it is cleared first. If the objects are NULL, the error indicator is cleared. Do not pass a NULL type and non-NULL value or traceback. The exception type should be a class. Do not pass an invalid exception type or value. (Violating these rules will cause subtle problems later.) This call takes away a reference to each object: you must own a reference to each object before the
call and after the call you no longer own these references. (If you don’t understand this, don’t use this function. I warned you.)

**Note:** This function is normally only used by code that needs to save and restore the error indicator temporarily; use `PyErr_Fetch()` to save the current exception state.

void **PyErr_SetString**(PyObject *type, const char *message)

This is the most common way to set the error indicator. The first argument specifies the exception type; it is normally one of the standard exceptions, e.g. `PyExc_RuntimeError`. You need not increment its reference count. The second argument is an error message; it is converted to a string object.

void **PyErr_SetObject**(PyObject *type, PyObject *value)

This function is similar to `PyErr_SetString()` but lets you specify an arbitrary Python object for the “value” of the exception.

PyObject* **PyErr_Format**(PyObject *exception, const char *format, ...)

*Return value: Always NULL.*

This function sets the error indicator and returns `NULL`. `exception` should be a Python exception class. The `format` and subsequent parameters help format the error message; they have the same meaning and values as in `PyString_FromFormat()`.

void **PyErr_SetNone**(PyObject *type)

This is a shorthand for `PyErr_SetObject(type, Py_None)`.

int **PyErr_BadArgument**()

This is a shorthand for `PyErr_SetString(PyExc_TypeError, message)`, where `message` indicates that a built-in operation was invoked with an illegal argument. It is mostly for internal use.
PyObject* PyErr_NoMemory()

Return value: Always NULL.
This is a shorthand for PyErr_SetNone(PyExc_MemoryError); it returns NULL so an object allocation function can write return PyErr_NoMemory(); when it runs out of memory.

PyObject* PyErr_SetFromErrno(PyObject *type)

Return value: Always NULL.
This is a convenience function to raise an exception when a C library function has returned an error and set the C variable errno. It constructs a tuple object whose first item is the integer errno value and whose second item is the corresponding error message (gotten from strerror()), and then calls PyErr_SetObject(type, object). On Unix, when the errno value is EINTR, indicating an interrupted system call, this calls PyErr_CheckSignals(), and if that set the error indicator, leaves it set to that. The function always returns NULL, so a wrapper function around a system call can write return PyErr_SetFromErrno(type); when the system call returns an error.

PyObject* PyErr_SetFromErrnoWithFilenameObject(PyObject *type, PyObject *filenameObject)

Similar to PyErr_SetFromErrno(), with the additional behavior that if filenameObject is not NULL, it is passed to the constructor of type as a third parameter. In the case of exceptions such as IOError and OSError, this is used to define the filename attribute of the exception instance.

PyObject* PyErr_SetFromErrnoWithFilename(PyObject *type, const char *filename)

Return value: Always NULL.
Similar to PyErr_SetFromErrnoWithFilenameObject(), but the
filename is given as a C string.

PyObject* PyErr_SetFromWindowsErr(int ierr)

*Return value:* Always **NULL**.

This is a convenience function to raise **WindowsError**. If called with `ierr` of 0, the error code returned by a call to `GetLastError()` is used instead. It calls the Win32 function `FormatMessage()` to retrieve the Windows description of error code given by `ierr` or `GetLastError()`, then it constructs a tuple object whose first item is the `ierr` value and whose second item is the corresponding error message (gotten from `FormatMessage()`), and then calls `PyErr_SetObject(PyExc_WindowsError, object)`. This function always returns **NULL**. Availability: Windows.

PyObject* PyErr_SetExcFromWindowsErr(PyObject *type, int ierr)

*Return value:* Always **NULL**.

Similar to `PyErr_SetFromWindowsErr()`, with an additional parameter specifying the exception type to be raised. Availability: Windows.

*New in version 2.3.*

PyObject* PyErr_SetFromWindowsErrWithFilenameObject(int ierr, PyObject *filenameObject)

Similar to `PyErr_SetFromWindowsErr()`, with the additional behavior that if `filenameObject` is not **NULL**, it is passed to the constructor of **WindowsError** as a third parameter. Availability: Windows.

PyObject* PyErr_SetFromWindowsErrWithFilename(int ierr, const char *filename)

*Return value:* Always **NULL**.

Similar to `PyErr_SetFromWindowsErrWithFilenameObject()`, but the filename is given as a C string. Availability: Windows.
PyObject*
PyErr_SetExcFromWindowsErrWithFilenameObject(PyObject *type, int ierr, PyObject *filename)

Similar to PyErr_SetFromWindowsErrWithFilenameObject(), with an additional parameter specifying the exception type to be raised. Availability: Windows.

*New in version 2.3.*

PyObject*
PyErr_SetExcFromWindowsErrWithFilename(PyObject *type, int ierr, const char *filename)

*Return value: Always NULL.*

Similar to PyErr_SetFromWindowsErrWithFilename(), with an additional parameter specifying the exception type to be raised. Availability: Windows.

*New in version 2.3.*

void PyErr_BadInternalCall()

This is a shorthand for PyErr_SetString(PyExc_SystemError, message), where message indicates that an internal operation (e.g. a Python/C API function) was invoked with an illegal argument. It is mostly for internal use.

int PyErr_WarnEx(PyObject *category, char *message, int stacklevel)

Issue a warning message. The category argument is a warning category (see below) or NULL; the message argument is a message string. stacklevel is a positive number giving a number of stack frames; the warning will be issued from the currently executing line of code in that stack frame. A stacklevel of 1 is the function calling PyErr_WarnEx(), 2 is the function above that, and so forth.
This function normally prints a warning message to `sys.stderr`; however, it is also possible that the user has specified that warnings are to be turned into errors, and in that case this will raise an exception. It is also possible that the function raises an exception because of a problem with the warning machinery (the implementation imports the `warnings` module to do the heavy lifting). The return value is 0 if no exception is raised, or -1 if an exception is raised. (It is not possible to determine whether a warning message is actually printed, nor what the reason is for the exception; this is intentional.) If an exception is raised, the caller should do its normal exception handling (for example, `Py_DECREF()` owned references and return an error value).

Warning categories must be subclasses of `Warning`; the default warning category is `RuntimeWarning`. The standard Python warning categories are available as global variables whose names are `PyExc_` followed by the Python exception name. These have the type `PyObject*`; they are all class objects. Their names are `PyExc_Warning`, `PyExc_UserWarning`, `PyExc_UnicodeWarning`, `PyExc_DeprecationWarning`, `PyExc_SyntaxWarning`, `PyExc_RuntimeWarning`, and `PyExc_FutureWarning`. `PyExc_Warning` is a subclass of `PyExc_Exception`; the other warning categories are subclasses of `PyExc_Warning`.

For information about warning control, see the documentation for the `warnings` module and the `-W` option in the command line documentation. There is no C API for warning control.

```c
int PyErr_Warn(PyObject *category, char *message)
```
Issue a warning message. The `category` argument is a warning category (see below) or `NULL`; the `message` argument is a message string. The warning will appear to be issued from the function calling `PyErr_Warn()`, equivalent to calling
PyErr_WarnEx() with a stacklevel of 1.

Deprecated; use PyErr_WarnEx() instead.

int PyErr_WARNExplicit(PyObject *category, const char *message, const char *filename, int lineno, const char *module, PyObject *registry)

Issue a warning message with explicit control over all warning attributes. This is a straightforward wrapper around the Python function warnings.warn_explicit(), see there for more information. The module and registry arguments may be set to NULL to get the default effect described there.

int PyErr_WARNPy3k(char *message, int stacklevel)

Issue a DeprecationWarning with the given message and stacklevel if the Py_Py3kWarningFlag flag is enabled.

New in version 2.6.

int PyErr_CheckSignals()

This function interacts with Python’s signal handling. It checks whether a signal has been sent to the processes and if so, invokes the corresponding signal handler. If the signal module is supported, this can invoke a signal handler written in Python. In all cases, the default effect for SIGINT is to raise the KeyboardInterrupt exception. If an exception is raised the error indicator is set and the function returns -1; otherwise the function returns 0. The error indicator may or may not be cleared if it was previously set.

void PyErr_SetInterrupt()

This function simulates the effect of a SIGINT signal arriving — the next time PyErr_CheckSignals() is called, KeyboardInterrupt will be raised. It may be called without holding the interpreter
int PySignal_SetWakeupFd(int fd)

This utility function specifies a file descriptor to which a '0' byte will be written whenever a signal is received. It returns the previous such file descriptor. The value -1 disables the feature; this is the initial state. This is equivalent to signal.set_wakeup_fd() in Python, but without any error checking. fd should be a valid file descriptor. The function should only be called from the main thread.

New in version 2.6.

PyObject* PyErr_NewException(char *name, PyObject *base, PyObject *dict)

This utility function creates and returns a new exception class. The name argument must be the name of the new exception, a C string of the form module.classname. The base and dict arguments are normally NULL. This creates a class object derived from Exception (accessible in C as PyExc_Exception).

The __module__ attribute of the new class is set to the first part (up to the last dot) of the name argument, and the class name is set to the last part (after the last dot). The base argument can be used to specify alternate base classes; it can either be only one class or a tuple of classes. The dict argument can be used to specify a dictionary of class variables and methods.

PyObject* PyErr_NewExceptionWithDoc(char *name, char *doc, PyObject *base, PyObject *dict)

Same as PyErr_NewException(), except that the new exception class can easily be given a docstring: If doc is non-NULL, it will be used as the docstring for the exception class.
New in version 2.7.

void PyErr_WriteUnraisable(PyObject *obj)

This utility function prints a warning message to sys.stderr when an exception has been set but it is impossible for the interpreter to actually raise the exception. It is used, for example, when an exception occurs in an _del__() method.

The function is called with a single argument obj that identifies the context in which the unraisable exception occurred. If possible, the repr of obj will be printed in the warning message.
Unicode Exception Objects

The following functions are used to create and modify Unicode exceptions from C.

**PyObject** * PyUnicodeDecodeError_Create(const char *encoding, const char *object, Py_ssize_t length, Py_ssize_t start, Py_ssize_t end, const char *reason)

Create a `UnicodeDecodeError` object with the attributes `encoding`, `object`, `length`, `start`, `end` and `reason`.

**PyObject** * PyUnicodeEncodeError_Create(const char *encoding, const Py_UNICODE *object, Py_ssize_t length, Py_ssize_t start, Py_ssize_t end, const char *reason)

Create a `UnicodeEncodeError` object with the attributes `encoding`, `object`, `length`, `start`, `end` and `reason`.

**PyObject** * PyUnicodeTranslateError_Create(const Py_UNICODE *object, Py_ssize_t length, Py_ssize_t start, Py_ssize_t end, const char *reason)

Create a `UnicodeTranslateError` object with the attributes `object`, `length`, `start`, `end` and `reason`.

**PyObject** * PyUnicodeDecodeError_GetEncoding(PyObject *exc)

**PyObject** * PyUnicodeEncodeError_GetEncoding(PyObject *exc)

Return the `encoding` attribute of the given exception object.

**PyObject** * PyUnicodeDecodeError_GetObject(PyObject *exc)

**PyObject** * PyUnicodeEncodeError_GetObject(PyObject *exc)

**PyObject** * PyUnicodeTranslateError_GetObject(PyObject *exc)

Return the `object` attribute of the given exception object.

int PyUnicodeDecodeError_GetStart(PyObject *exc, Py_ssize_t *start)
int PyUnicodeEncodeError_GetStart(PyObject *exc,
Py_ssize_t *start
int PyUnicodeTranslateError_GetStart(PyObject *exc, Py_ssize_t *start)
    Get the start attribute of the given exception object and place it into *start. start must not be NULL. Return 0 on success, -1 on failure.

int PyUnicodeDecodeError_SetStart(PyObject *exc, Py_ssize_t start)
int PyUnicodeEncodeError_SetStart(PyObject *exc, Py_ssize_t start)
int PyUnicodeTranslateError_SetStart(PyObject *exc, Py_ssize_t start)
    Set the start attribute of the given exception object to start. Return 0 on success, -1 on failure.

int PyUnicodeDecodeError_GetEnd(PyObject *exc, Py_ssize_t *end)
int PyUnicodeEncodeError_GetEnd(PyObject *exc, Py_ssize_t *end)
int PyUnicodeTranslateError_GetEnd(PyObject *exc, Py_ssize_t *end)
    Get the end attribute of the given exception object and place it into *end. end must not be NULL. Return 0 on success, -1 on failure.

int PyUnicodeDecodeError_SetEnd(PyObject *exc, Py_ssize_t end)
int PyUnicodeEncodeError_SetEnd(PyObject *exc, Py_ssize_t end)
int PyUnicodeTranslateError_SetEnd(PyObject *exc, Py_ssize_t end)
    Set the end attribute of the given exception object to end. Return 0 on success, -1 on failure.

PyObject* PyUnicodeDecodeError_GetReason(PyObject *exc)
PyObject* PyUnicodeEncodeError_GetReason(PyObject *exc)
PyObject* PyUnicodeTranslateError_GetReason(PyObject *exc)
Return the *reason* attribute of the given exception object.

```c
int PyUnicodeDecodeError_SetReason(PyObject *exc, const char *reason)

int PyUnicodeEncodeError_SetReason(PyObject *exc, const char *reason)

int PyUnicodeTranslateError_SetReason(PyObject *exc, const char *reason)
```

Set the *reason* attribute of the given exception object to *reason*. Return 0 on success, -1 on failure.
Recursion Control

These two functions provide a way to perform safe recursive calls at the C level, both in the core and in extension modules. They are needed if the recursive code does not necessarily invoke Python code (which tracks its recursion depth automatically).

```c
int Py_EnterRecursiveCall(const char *where)
```

Marks a point where a recursive C-level call is about to be performed.

If `USE_STACKCHECK` is defined, this function checks if the OS stack overflowed using `PyOS_CheckStack()`. In this case, it sets a `MemoryError` and returns a nonzero value.

The function then checks if the recursion limit is reached. If this is the case, a `RuntimeError` is set and a nonzero value is returned. Otherwise, zero is returned.

*where* should be a string such as " in instance check" to be concatenated to the `RuntimeError` message caused by the recursion depth limit.

```c
void Py_LeaveRecursiveCall()
```

Ends a `Py_EnterRecursiveCall()`. Must be called once for each successful invocation of `Py_EnterRecursiveCall()`. 

**Standard Exceptions**

All standard Python exceptions are available as global variables whose names are `PyExc_` followed by the Python exception name. These have the type `PyObject*`; they are all class objects. For completeness, here are all the variables:

<table>
<thead>
<tr>
<th>C Name</th>
<th>Python Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyExc_BaseException</td>
<td>BaseException</td>
<td>(1),(4)</td>
</tr>
<tr>
<td>PyExc_Exception</td>
<td>Exception</td>
<td>(1)</td>
</tr>
<tr>
<td>PyExc_StandardError</td>
<td>StandardError</td>
<td>(1)</td>
</tr>
<tr>
<td>PyExc_ArithmeticError</td>
<td>ArithmeticError</td>
<td>(1)</td>
</tr>
<tr>
<td>PyExc_LookupError</td>
<td>LookupError</td>
<td>(1)</td>
</tr>
<tr>
<td>PyExc_AssertionError</td>
<td>AssertionError</td>
<td></td>
</tr>
<tr>
<td>PyExc_AttributeError</td>
<td>AttributeError</td>
<td></td>
</tr>
<tr>
<td>PyExc_EOFError</td>
<td>EOFError</td>
<td></td>
</tr>
<tr>
<td>PyExc_EnvironmentError</td>
<td>EnvironmentError</td>
<td>(1)</td>
</tr>
<tr>
<td>PyExc_FloatingPointError</td>
<td>FloatingPointError</td>
<td></td>
</tr>
<tr>
<td>PyExc_IOError</td>
<td>IOError</td>
<td></td>
</tr>
<tr>
<td>PyExc_ImportError</td>
<td>ImportError</td>
<td></td>
</tr>
<tr>
<td>PyExc_IndexError</td>
<td>IndexError</td>
<td></td>
</tr>
<tr>
<td>PyExc_KeyError</td>
<td>KeyError</td>
<td></td>
</tr>
<tr>
<td>PyExc_KeyboardInterrupt</td>
<td>KeyboardInterrupt</td>
<td></td>
</tr>
<tr>
<td>PyExc_MemoryError</td>
<td>MemoryError</td>
<td></td>
</tr>
<tr>
<td>PyExc_NameError</td>
<td>NameError</td>
<td></td>
</tr>
<tr>
<td>PyExc_NotImplementedError</td>
<td>NotImplementedError</td>
<td></td>
</tr>
<tr>
<td>PyExc_OSError</td>
<td>OSError</td>
<td></td>
</tr>
<tr>
<td>PyExc_OverflowError</td>
<td>OverflowError</td>
<td></td>
</tr>
<tr>
<td>PyExc_ReferenceError</td>
<td>ReferenceError</td>
<td></td>
</tr>
<tr>
<td>PyExc_RuntimeError</td>
<td>RuntimeError</td>
<td></td>
</tr>
<tr>
<td>PyExc_SyntaxError</td>
<td>SyntaxError</td>
<td></td>
</tr>
<tr>
<td>Exception Class</td>
<td>Exception Name</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>PyExc_SystemError</td>
<td>SystemError</td>
<td></td>
</tr>
<tr>
<td>PyExc_SystemExit</td>
<td>SystemExit</td>
<td></td>
</tr>
<tr>
<td>PyExc_TypeError</td>
<td>TypeError</td>
<td></td>
</tr>
<tr>
<td>PyExc_ValueError</td>
<td>ValueError</td>
<td></td>
</tr>
<tr>
<td>PyExc_WindowsError</td>
<td>WindowsError</td>
<td>(3)</td>
</tr>
<tr>
<td>PyExc_ZeroDivisionError</td>
<td>ZeroDivisionError</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. This is a base class for other standard exceptions.

2. This is the same as `weakref.ReferenceError`.

3. Only defined on Windows; protect code that uses this by testing that the preprocessor macro `MS_WINDOWS` is defined.

4. *New in version 2.5.*
String Exceptions

*Changed in version 2.6:* All exceptions to be raised or caught must be derived from `BaseException`. Trying to raise a string exception now raises `TypeError`. 
The functions in this chapter perform various utility tasks, ranging from helping C code be more portable across platforms, using Python modules from C, and parsing function arguments and constructing Python values from C values.

- Operating System Utilities
- System Functions
- Process Control
- Importing Modules
- Data marshalling support
- Parsing arguments and building values
- String conversion and formatting
- Reflection
- Codec registry and support functions
  - Codec lookup API
  - Registry API for Unicode encoding error handlers
int Py_FdIsInteractive(FILE *fp, const char *filename)

Return true (nonzero) if the standard I/O file fp with name filename is deemed interactive. This is the case for files for which isatty(fileno(fp)) is true. If the global flag Py_InteractiveFlag is true, this function also returns true if the filename pointer is NULL or if the name is equal to one of the strings '<stdin>' or '???'.

void PyOS_AfterFork()

Function to update some internal state after a process fork; this should be called in the new process if the Python interpreter will continue to be used. If a new executable is loaded into the new process, this function does not need to be called.

int PyOS_CheckStack()

Return true when the interpreter runs out of stack space. This is a reliable check, but is only available when USE_STACKCHECK is defined (currently on Windows using the Microsoft Visual C++ compiler). USE_STACKCHECK will be defined automatically; you should never change the definition in your own code.

PyOS_sighandler_t PyOS_getsig(int i)

Return the current signal handler for signal i. This is a thin wrapper around either sigaction() or signal(). Do not call those functions directly! PyOS_sighandler_t is a typedef alias for void (*)(int).

PyOS_sighandler_t PyOS_setsig(int i, PyOS_sighandler_t h)

Set the signal handler for signal i to be h; return the old signal handler. This is a thin wrapper around either sigaction() or signal(). Do not call those functions directly! PyOS_sighandler_t
is a typedef alias for `void (*)(int)`.
System Functions

These are utility functions that make functionality from the `sys` module accessible to C code. They all work with the current interpreter thread’s `sys` module’s dict, which is contained in the internal thread state structure.

`PyObject *PySys_GetObject(char *name)`

*Return value: Borrowed reference.*
Return the object `name` from the `sys` module or `NULL` if it does not exist, without setting an exception.

`FILE *PySys_GetFile(char *name, FILE *def)`

Return the `FILE*` associated with the object `name` in the `sys` module, or `def` if `name` is not in the module or is not associated with a `FILE*`.

`int PySys_SetObject(char *name, PyObject *v)`

Set `name` in the `sys` module to `v` unless `v` is `NULL`, in which case `name` is deleted from the sys module. Returns `0` on success, `-1` on error.

`void PySys_ResetWarnOptions()`

Reset `sys.warnoptions` to an empty list.

`void PySys_AddWarnOption(char *s)`

Append `s` to `sys.warnoptions`.

`void PySys_SetPath(char *path)`

Set `sys.path` to a list object of paths found in `path` which should be a list of paths separated with the platform’s search path delimiter (`:` on Unix, `;` on Windows).
void PySys_WriteStdout(const char *format, ...)
Write the output string described by format to sys.stdout. No exceptions are raised, even if truncation occurs (see below).

format should limit the total size of the formatted output string to 1000 bytes or less – after 1000 bytes, the output string is truncated. In particular, this means that no unrestricted “%s” formats should occur; these should be limited using “%.<N>s” where <N> is a decimal number calculated so that <N> plus the maximum size of other formatted text does not exceed 1000 bytes. Also watch out for “%f”, which can print hundreds of digits for very large numbers.

If a problem occurs, or sys.stdout is unset, the formatted message is written to the real (C level) stdout.

void PySys_WriteStderr(const char *format, ...)
As above, but write to sys.stderr or stderr instead.
void Py_FatalError(const char *message)

Print a fatal error message and kill the process. No cleanup is performed. This function should only be invoked when a condition is detected that would make it dangerous to continue using the Python interpreter; e.g., when the object administration appears to be corrupted. On Unix, the standard C library function abort() is called which will attempt to produce a core file.

void Py_Exit(int status)

Exit the current process. This calls Py_Finalize() and then calls the standard C library function exit(status).

int Py_AtExit(void (*func)())

Register a cleanup function to be called by Py_Finalize(). The cleanup function will be called with no arguments and should return no value. At most 32 cleanup functions can be registered. When the registration is successful, Py_AtExit() returns 0; on failure, it returns -1. The cleanup function registered last is called first. Each cleanup function will be called at most once. Since Python's internal finalization will have completed before the cleanup function, no Python APIs should be called by func.
Importing Modules

**PyObject** `PyImport_ImportModule`(const char *name)

*Return value: New reference.*

This is a simplified interface to `PyImport_ImportModuleEx()` below, leaving the `globals` and `locals` arguments set to `NULL` and `level` set to 0. When the `name` argument contains a dot (when it specifies a submodule of a package), the `fromlist` argument is set to the list `['*']` so that the return value is the named module rather than the top-level package containing it as would otherwise be the case. (Unfortunately, this has an additional side effect when `name` in fact specifies a subpackage instead of a submodule: the submodules specified in the package’s `__all__` variable are loaded.) Return a new reference to the imported module, or `NULL` with an exception set on failure. Before Python 2.4, the module may still be created in the failure case — examine `sys.modules` to find out. Starting with Python 2.4, a failing import of a module no longer leaves the module in `sys.modules`.

*Changed in version 2.4:* Failing imports remove incomplete module objects.

*Changed in version 2.6:* Always uses absolute imports.

**PyObject** `PyImport_ImportModuleNoBlock`(const char *name)

This version of `PyImport_ImportModule()` does not block. It’s intended to be used in C functions that import other modules to execute a function. The import may block if another thread holds the import lock. The function `PyImport_ImportModuleNoBlock()` never blocks. It first tries to fetch the module from `sys.modules` and falls back to `PyImport_ImportModule()` unless the lock is held, in which case the function will raise an `ImportError`. 
New in version 2.6.

PyObject* PyImport_ImportModuleEx(char *name,  
PyObject *globals, PyObject *locals, PyObject *fromlist)

Import a module. This is best described by referring to the built-in Python function __import__() as the standard __import__() function calls this function directly.

The return value is a new reference to the imported module or top-level package, or NULL with an exception set on failure (before Python 2.4, the module may still be created in this case). Like for __import__(), the return value when a submodule of a package was requested is normally the top-level package, unless a non-empty fromlist was given.

Changed in version 2.4: Failing imports remove incomplete module objects.

Changed in version 2.6: The function is an alias for PyImport_ImportModuleLevel() with -1 as level, meaning relative import.

PyObject* PyImport_ImportModuleLevel(char *name,  
PyObject *globals, PyObject *locals, PyObject *fromlist, int level)

Import a module. This is best described by referring to the built-in Python function __import__() as the standard __import__() function calls this function directly.

The return value is a new reference to the imported module or top-level package, or NULL with an exception set on failure. Like for __import__(), the return value when a submodule of a package was requested is normally the top-level package, unless a non-empty fromlist was given.
New in version 2.5.

PyObject* PyImport_Import(PyObject *name)

This is a higher-level interface that calls the current “import hook function”. It invokes the __import__() function from the __builtins__ of the current globals. This means that the import is done using whatever import hooks are installed in the current environment, e.g. by rexec or ihooks.

Changed in version 2.6: Always uses absolute imports.

PyObject* PyImport_ReloadModule(PyObject *m)

Reload a module. This is best described by referring to the built-in Python function reload(), as the standard reload() function calls this function directly. Return a new reference to the reloaded module, or NULL with an exception set on failure (the module still exists in this case).

PyObject* PyImport_AddModule(const char *name)

Return value: Borrowed reference.
Return the module object corresponding to a module name. The name argument may be of the form package.module. First check the modules dictionary if there’s one there, and if not, create a new one and insert it in the modules dictionary. Return NULL with an exception set on failure.

Note: This function does not load or import the module; if the module wasn’t already loaded, you will get an empty module object. Use PyImport_ImportModule() or one of its variants to import a module. Package structures implied by a dotted name for name are not created if not already present.
PyObject* PyImport_ExecCodeModule(char *name, PyObject *co)

*Return value: New reference.*

Given a module name (possibly of the form `package.module`) and a code object read from a Python bytecode file or obtained from the built-in function `compile()`, load the module. Return a new reference to the module object, or `NULL` with an exception set if an error occurred. Before Python 2.4, the module could still be created in error cases. Starting with Python 2.4, `name` is removed from `sys.modules` in error cases, and even if `name` was already in `sys.modules` on entry to `PyImport_ExecCodeModule()`. Leaving incompletely initialized modules in `sys.modules` is dangerous, as imports of such modules have no way to know that the module object is an unknown (and probably damaged with respect to the module author’s intents) state.

The module’s `__file__` attribute will be set to the code object’s `co_filename`.

This function will reload the module if it was already imported. See `PyImport_ReloadModule()` for the intended way to reload a module.

If `name` points to a dotted name of the form `package.module`, any package structures not already created will still not be created.

*Changed in version 2.4: name is removed from sys.modules in error cases.*

PyObject* PyImport_ExecCodeModuleEx(char *name, PyObject *co, char *pathname)

*Return value: New reference.*

Like `PyImport_ExecCodeModule()`, but the `__file__` attribute of the module object is set to `pathname` if it is non-`NULL`. 
long PyImport_GetMagicNumber()
    Return the magic number for Python bytecode files (a.k.a. .pyc and .pyo files). The magic number should be present in the first four bytes of the bytecode file, in little-endian byte order.

PyObject* PyImport_GetModuleDict()
    Return value: Borrowed reference.
    Return the dictionary used for the module administration (a.k.a. sys.modules). Note that this is a per-interpreter variable.

PyObject* PyImport_GetImporter(PyObject *path)
    Return an importer object for a sys.path/pkg.__path__ item path, possibly by fetching it from the sys.path_importer_cache dict. If it wasn’t yet cached, traverse sys.path_hooks until a hook is found that can handle the path item. Return None if no hook could; this tells our caller it should fall back to the built-in import mechanism. Cache the result in sys.path_importer_cache. Return a new reference to the importer object.

    New in version 2.6.

void _PyImport_Init()
    Initialize the import mechanism. For internal use only.

void PyImport_Cleanup()
    Empty the module table. For internal use only.

void _PyImport_Fini()
    Finalize the import mechanism. For internal use only.

PyObject* _PyImport_FindExtension(char *, char *)
    For internal use only.

PyObject* _PyImport_FixupExtension(char *, char *)
    For internal use only.
int PyImport_ImportFrozenModule(char *name)
    Load a frozen module named name. Return 1 for success, 0 if the module is not found, and -1 with an exception set if the initialization failed. To access the imported module on a successful load, use PyImport_ImportModule(). (Note the misnomer — this function would reload the module if it was already imported.)

struct _frozen
    This is the structure type definition for frozen module descriptors, as generated by the freeze utility (see Tools/freeze/ in the Python source distribution). Its definition, found in Include/import.h, is:

```
struct _frozen {
    char *name;
    unsigned char *code;
    int size;
};
```

struct _frozen* PyImport_FrozenModules
    This pointer is initialized to point to an array of struct _frozen records, terminated by one whose members are all NULL or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

int PyImport_AppendInittab(const char *name, void (*initfunc)(void))
    Add a single module to the existing table of built-in modules. This is a convenience wrapper around PyImport_ExtendInittab(), returning -1 if the table could not be extended. The new module can be imported by the name name, and uses the function initfunc as the initialization function called on the first attempted import. This should be called before Py_Initialize().
struct _inittab

Structure describing a single entry in the list of built-in modules. Each of these structures gives the name and initialization function for a module built into the interpreter. Programs which embed Python may use an array of these structures in conjunction with PyImport_ExtendInittab() to provide additional built-in modules. The structure is defined in `Include/import.h` as:

```c
struct _inittab {
    char *name;
    void (*initfunc)(void);
};
```

int PyImport_ExtendInittab(struct _inittab *newtab)

Add a collection of modules to the table of built-in modules. The `newtab` array must end with a sentinel entry which contains `NULL` for the `name` field; failure to provide the sentinel value can result in a memory fault. Returns 0 on success or -1 if insufficient memory could be allocated to extend the internal table. In the event of failure, no modules are added to the internal table. This should be called before `Py_Initialize()`. 
Data marshalling support

These routines allow C code to work with serialized objects using the same data format as the `marshal` module. There are functions to write data into the serialization format, and additional functions that can be used to read the data back. Files used to store marshalled data must be opened in binary mode.

Numeric values are stored with the least significant byte first.

The module supports two versions of the data format: version 0 is the historical version, version 1 (new in Python 2.4) shares interned strings in the file, and upon unmarshalling. Version 2 (new in Python 2.5) uses a binary format for floating point numbers. `Py_MARSHAL_VERSION` indicates the current file format (currently 2).

```c
void PyMarshal_WriteLongToFile(long value, FILE *file, int version)
    Marshal a long integer, value, to file. This will only write the least-significant 32 bits of value; regardless of the size of the native long type.

    Changed in version 2.4: version indicates the file format.
```

```c
void PyMarshal_WriteObjectToFile(PyObject *value, FILE *file, int version)
    Marshal a Python object, value, to file.

    Changed in version 2.4: version indicates the file format.
```

```c
PyObject* PyMarshal_WriteObjectToString(PyObject *value, int version)
    Return a string object containing the marshalled representation of
```
value.

*Changed in version 2.4*: `version` indicates the file format.

The following functions allow marshalled values to be read back in.

XXX What about error detection? It appears that reading past the end of the file will always result in a negative numeric value (where that's relevant), but it's not clear that negative values won't be handled properly when there's no error. What's the right way to tell? Should only non-negative values be written using these routines?

```c
long PyMarshal_ReadLongFromFile(FILE *file)
    Return a C `long` from the data stream in a `FILE*` opened for reading. Only a 32-bit value can be read in using this function, regardless of the native size of `long`.
```

```c
int PyMarshal_ReadShortFromFile(FILE *file)
    Return a C `short` from the data stream in a `FILE*` opened for reading. Only a 16-bit value can be read in using this function, regardless of the native size of `short`.
```

```c
PyObject* PyMarshal_ReadObjectFromFile(FILE *file)
    Return a Python object from the data stream in a `FILE*` opened for reading. On error, sets the appropriate exception (`EOFError` or `TypeError`) and returns `NULL`.
```

```c
PyObject* PyMarshal_ReadLastObjectFromFile(FILE *file)
    Return a Python object from the data stream in a `FILE*` opened for reading. Unlike `PyMarshal_ReadObjectFromFile()`, this function assumes that no further objects will be read from the file, allowing it to aggressively load file data into memory so that the de-serialization can operate from data in memory rather than reading
a byte at a time from the file. Only use these variant if you are certain that you won’t be reading anything else from the file. On error, sets the appropriate exception (\texttt{EOFError} or \texttt{TypeError}) and returns \texttt{NULL}.

\textbf{PyObject*} \texttt{PyMarshal_ReadObjectFromString(char *string, Py_ssize_t len)}

\textit{Return value: New reference.}

Return a Python object from the data stream in a character buffer containing \texttt{len} bytes pointed to by \texttt{string}. On error, sets the appropriate exception (\texttt{EOFError} or \texttt{TypeError}) and returns \texttt{NULL}.

\textit{Changed in version 2.5:} This function used an \texttt{int} type for \texttt{len}. This might require changes in your code for properly supporting 64-bit systems.
Parsing arguments and building values

These functions are useful when creating your own extensions functions and methods. Additional information and examples are available in Extending and Embedding the Python Interpreter.

The first three of these functions described, `PyArg_ParseTuple()`, `PyArg_ParseTupleAndKeywords()`, and `PyArg_Parse()`, all use format strings which are used to tell the function about the expected arguments. The format strings use the same syntax for each of these functions.

A format string consists of zero or more “format units.” A format unit describes one Python object; it is usually a single character or a parenthesized sequence of format units. With a few exceptions, a format unit that is not a parenthesized sequence normally corresponds to a single address argument to these functions. In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that matches the format unit; and the entry in [square] brackets is the type of the C variable(s) whose address should be passed.

These formats allow accessing an object as a contiguous chunk of memory. You don’t have to provide raw storage for the returned unicode or bytes area. Also, you won’t have to release any memory yourself, except with the `es`, `es#`, `et` and `et#` formats.

`s` (string or Unicode) [const char *]
Convert a Python string or Unicode object to a C pointer to a character string. You must not provide storage for the string itself; a pointer to an existing string is stored into the character pointer variable whose address you pass. The C string is NUL-
terminated. The Python string must not contain embedded NULL bytes; if it does, a `TypeError` exception is raised. Unicode objects are converted to C strings using the default encoding. If this conversion fails, a `UnicodeError` is raised.

`s#` (string, Unicode or any read buffer compatible object) [const char *, int (or `Py_ssize_t`, see below)]

This variant on `s` stores into two C variables, the first one a pointer to a character string, the second one its length. In this case the Python string may contain embedded null bytes. Unicode objects pass back a pointer to the default encoded string version of the object if such a conversion is possible. All other read-buffer compatible objects pass back a reference to the raw internal data representation.

Starting with Python 2.5 the type of the length argument can be controlled by defining the macro `PY_SSIZE_T_CLEAN` before including `Python.h`. If the macro is defined, length is a `Py_ssize_t` rather than an int.

`s*` (string, Unicode, or any buffer compatible object) [Py_buffer]

Similar to `s#`, this code fills a Py_buffer structure provided by the caller. The buffer gets locked, so that the caller can subsequently use the buffer even inside a `Py_BEGIN_ALLOW_THREADS` block; the caller is responsible for calling `PyBuffer_Release` with the structure after it has processed the data.

*New in version 2.6.*

`z` (string, Unicode or `None`) [const char *]

Like `s`, but the Python object may also be `None`, in which case the C pointer is set to `NULL`.

`z#` (string, Unicode, `None` or any read buffer compatible object) [const char *, int]
This is to \texttt{s\#} as \texttt{z} is to \texttt{s}.

\texttt{z*} (string, Unicode, \texttt{None} or any buffer compatible object) [Py_buffer]

This is to \texttt{s*} as \texttt{z} is to \texttt{s}.

\textit{New in version 2.6.}

\texttt{u} (Unicode) [Py\_UNICODE \*]

Convert a Python Unicode object to a C pointer to a NUL-terminated buffer of 16-bit Unicode (UTF-16) data. As with \texttt{s}, there is no need to provide storage for the Unicode data buffer; a pointer to the existing Unicode data is stored into the \texttt{Py\_UNICODE} pointer variable whose address you pass.

\texttt{u\#} (Unicode) [Py\_UNICODE \*, int]

This variant on \texttt{u} stores into two C variables, the first one a pointer to a Unicode data buffer, the second one its length. Non-Unicode objects are handled by interpreting their read-buffer pointer as pointer to a \texttt{Py\_UNICODE} array.

\texttt{es} (string, Unicode or character buffer compatible object) [const char *encoding, char **buffer]

This variant on \texttt{s} is used for encoding Unicode and objects convertible to Unicode into a character buffer. It only works for encoded data without embedded NUL bytes.

This format requires two arguments. The first is only used as input, and must be a \texttt{const char*} which points to the name of an encoding as a NUL-terminated string, or \texttt{NULL}, in which case the default encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a \texttt{char**}; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument.

\texttt{PyArg\_ParseTuple()} will allocate a buffer of the needed size, copy
the encoded data into this buffer and adjust *buffer to reference the newly allocated storage. The caller is responsible for calling PyMem_Free() to free the allocated buffer after use.

et (string, Unicode or character buffer compatible object) [const char *encoding, char **buffer]
Same as es except that 8-bit string objects are passed through without recoding them. Instead, the implementation assumes that the string object uses the encoding passed in as parameter.

es# (string, Unicode or character buffer compatible object) [const char *encoding, char **buffer, int *buffer_length]
This variant on s# is used for encoding Unicode and objects convertible to Unicode into a character buffer. Unlike the es format, this variant allows input data which contains NUL characters.

It requires three arguments. The first is only used as input, and must be a const char* which points to the name of an encoding as a NUL-terminated string, or NULL, in which case the default encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a char**; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument. The third argument must be a pointer to an integer; the referenced integer will be set to the number of bytes in the output buffer.

There are two modes of operation:

If *buffer points a NULL pointer, the function will allocate a buffer of the needed size, copy the encoded data into this buffer and set *buffer to reference the newly allocated storage. The caller is responsible for calling PyMem_Free() to free the allocated buffer after usage.
If *buffer points to a non-NULL pointer (an already allocated buffer), PyArg_ParseTuple() will use this location as the buffer and interpret the initial value of *buffer_length as the buffer size. It will then copy the encoded data into the buffer and NUL-terminate it. If the buffer is not large enough, a TypeError will be set. Note: starting from Python 3.6 a ValueError will be set. In both cases, *buffer_length is set to the length of the encoded data without the trailing NUL byte.

et# (string, Unicode or character buffer compatible object) [const char *encoding, char **buffer, int *buffer_length]
Same as es# except that string objects are passed through without recoding them. Instead, the implementation assumes that the string object uses the encoding passed in as parameter.

b (integer) [unsigned char]
Convert a nonnegative Python integer to an unsigned tiny int, stored in a C unsigned char.

B (integer) [unsigned char]
Convert a Python integer to a tiny int without overflow checking, stored in a C unsigned char.

New in version 2.3.

h (integer) [short int]
Convert a Python integer to a C short int.

H (integer) [unsigned short int]
Convert a Python integer to a C unsigned short int, without overflow checking.

New in version 2.3.

i (integer) [int]
Convert a Python integer to a plain C \texttt{int}.

\texttt{I} (integer) [\texttt{unsigned int}]
Convert a Python integer to a C \texttt{unsigned int}, without overflow checking.

\textit{New in version 2.3.}

\texttt{l} (integer) [\texttt{long int}]
Convert a Python integer to a C \texttt{long int}.

\texttt{k} (integer) [\texttt{unsigned long}]
Convert a Python integer or long integer to a C \texttt{unsigned long} without overflow checking.

\textit{New in version 2.3.}

\texttt{L} (integer) [\texttt{PY_LONG_LONG}]
Convert a Python integer to a C \texttt{long long}. This format is only available on platforms that support \texttt{long long} (or \texttt{\_int64} on Windows).

\texttt{K} (integer) [\texttt{unsigned PY_LONG_LONG}]
Convert a Python integer or long integer to a C \texttt{unsigned long long} without overflow checking. This format is only available on platforms that support \texttt{unsigned long long} (or \texttt{unsigned \_int64} on Windows).

\textit{New in version 2.3.}

\texttt{n} (integer) [\texttt{Py_ssize_t}]
Convert a Python integer or long integer to a C \texttt{Py_ssize_t}.

\textit{New in version 2.5.}

\texttt{c} (string of length 1) [\texttt{char}]
Convert a Python character, represented as a string of length 1,
to a C char.

f (float) [float]
    Convert a Python floating point number to a C float.

d (float) [double]
    Convert a Python floating point number to a C double.

p (complex) [Py_complex]
    Convert a Python complex number to a C Py_complex structure.

0 (object) [PyObject *]
    Store a Python object (without any conversion) in a C object pointer. The C program thus receives the actual object that was passed. The object’s reference count is not increased. The pointer stored is not NULL.

0! (object) [typeobject, PyObject *]
    Store a Python object in a C object pointer. This is similar to 0, but takes two C arguments: the first is the address of a Python type object, the second is the address of the C variable (of type PyObject*) into which the object pointer is stored. If the Python object does not have the required type, TypeError is raised.

0& (object) [converter, anything]
    Convert a Python object to a C variable through a converter function. This takes two arguments: the first is a function, the second is the address of a C variable (of arbitrary type), converted to void *.

    status = converter(object, address);

where object is the Python object to be converted and address is the void* argument that was passed to the PyArg_Parse*() function. The returned status should be 1 for a successful conversion and 0 if the conversion has failed. When the
conversion fails, the converter function should raise an exception and leave the content of address unmodified.

\texttt{s} (string) [PyStringObject *]
Like o but requires that the Python object is a string object. Raises \texttt{TypeError} if the object is not a string object. The C variable may also be declared as \texttt{PyObject *}.

\texttt{u} (Unicode string) [PyUnicodeObject *]
Like o but requires that the Python object is a Unicode object. Raises \texttt{TypeError} if the object is not a Unicode object. The C variable may also be declared as \texttt{PyObject *}.

\texttt{t\#} (read-only character buffer) [char *, int]
Like \texttt{s\#}, but accepts any object which implements the read-only buffer interface. The char* variable is set to point to the first byte of the buffer, and the int is set to the length of the buffer. Only single-segment buffer objects are accepted; \texttt{TypeError} is raised for all others.

\texttt{w} (read-write character buffer) [char *]
Similar to \texttt{s}, but accepts any object which implements the read-write buffer interface. The caller must determine the length of the buffer by other means, or use \texttt{w\#} instead. Only single-segment buffer objects are accepted; \texttt{TypeError} is raised for all others.

\texttt{w\#} (read-write character buffer) [char *, Py_ssize_t]
Like \texttt{s\#}, but accepts any object which implements the read-write buffer interface. The char * variable is set to point to the first byte of the buffer, and the Py_ssize_t is set to the length of the buffer. Only single-segment buffer objects are accepted; \texttt{TypeError} is raised for all others.

\texttt{w*} (read-write byte-oriented buffer) [Py_buffer]
This is to \texttt{w} what \texttt{s*} is to \texttt{s}. 

New in version 2.6.

(items) (tuple) [matching-items]

The object must be a Python sequence whose length is the number of format units in items. The C arguments must correspond to the individual format units in items. Format units for sequences may be nested.

Note: Prior to Python version 1.5.2, this format specifier only accepted a tuple containing the individual parameters, not an arbitrary sequence. Code which previously caused TypeError to be raised here may now proceed without an exception. This is not expected to be a problem for existing code.

It is possible to pass Python long integers where integers are requested; however no proper range checking is done — the most significant bits are silently truncated when the receiving field is too small to receive the value (actually, the semantics are inherited from downcasts in C — your mileage may vary).

A few other characters have a meaning in a format string. These may not occur inside nested parentheses. They are:

\|

Indicates that the remaining arguments in the Python argument list are optional. The C variables corresponding to optional arguments should be initialized to their default value — when an optional argument is not specified, PyArg_ParseTuple() does not touch the contents of the corresponding C variable(s).

; The list of format units ends here; the string after the colon is used as the function name in error messages (the “associated value” of the exception that PyArg_ParseTuple() raises).
The list of format units ends here; the string after the semicolon is used as the error message instead of the default error message. : and ; mutually exclude each other.

Note that any Python object references which are provided to the caller are borrowed references; do not decrement their reference count!

Additional arguments passed to these functions must be addresses of variables whose type is determined by the format string; these are used to store values from the input tuple. There are a few cases, as described in the list of format units above, where these parameters are used as input values; they should match what is specified for the corresponding format unit in that case.

For the conversion to succeed, the arg object must match the format and the format must be exhausted. On success, the PyArg_Parse*() functions return true, otherwise they return false and raise an appropriate exception. When the PyArg_Parse*() functions fail due to conversion failure in one of the format units, the variables at the addresses corresponding to that and the following format units are left untouched.

int PyArg_ParseTuple(PyObject *args, const char *format, ...)
   Parse the parameters of a function that takes only positional parameters into local variables. Returns true on success; on failure, it returns false and raises the appropriate exception.

int PyArg_VaParse(PyObject *args, const char *format, va_list vargs)
   Identical to PyArg_ParseTuple(), except that it accepts a va_list rather than a variable number of arguments.

int PyArg_ParseTupleAndKeywords(PyObject *args, PyObject *kw, const char *format, char *keywords[], ...)
Parse the parameters of a function that takes both positional and keyword parameters into local variables. Returns true on success; on failure, it returns false and raises the appropriate exception.

```c
int PyArg_ParseTupleAndKeywords(PyObject *args, PyObject *kw, const char *format, char *keywords[], va_list vargs)

Identical to PyArg_ParseTupleAndKeywords(), except that it accepts a va_list rather than a variable number of arguments.
```

```c
int PyArg_Parse(PyObject *args, const char *format, ...)  

Function used to deconstruct the argument lists of “old-style” functions — these are functions which use the METH_OLDARGS parameter parsing method. This is not recommended for use in parameter parsing in new code, and most code in the standard interpreter has been modified to no longer use this for that purpose. It does remain a convenient way to decompose other tuples, however, and may continue to be used for that purpose.
```

```c
int PyArg_UnpackTuple(PyObject *args, const char *name, Py_ssize_t min, Py_ssize_t max, ...)  

A simpler form of parameter retrieval which does not use a format string to specify the types of the arguments. Functions which use this method to retrieve their parameters should be declared as METH_VARARGS in function or method tables. The tuple containing the actual parameters should be passed as args; it must actually be a tuple. The length of the tuple must be at least min and no more than max; min and max may be equal. Additional arguments must be passed to the function, each of which should be a pointer to a PyObject* variable; these will be filled in with the values from args; they will contain borrowed references. The variables which correspond to optional parameters not given by args will not be filled in; these should be initialized by the caller. This function returns true on success and false if args is not a
tuple or contains the wrong number of elements; an exception will be set if there was a failure.

This is an example of the use of this function, taken from the sources for the _weakref helper module for weak references:

```c
static PyObject *
weakref_ref(PyObject *self, PyObject *args)
{
    PyObject *object;
    PyObject *callback = NULL;
    PyObject *result = NULL;

    if (PyArg_UnpackTuple(args, "ref", 1, 2, &object, &callback))
        result = PyWeakref_NewRef(object, callback);

    return result;
}
```

The call to `PyArg_UnpackTuple()` in this example is entirely equivalent to this call to `PyArg_ParseTuple()`:

```c
PyArg_ParseTuple(args, "O|O:ref", &object, &callback)
```

*New in version 2.2.*

*Changed in version 2.5:* This function used an `int` type for `min` and `max`. This might require changes in your code for properly supporting 64-bit systems.

**PyObject** `Py_BuildValue` (const char *format, ...)

*Return value: New reference.*

Create a new value based on a format string similar to those accepted by the `PyArg_Parse*()` family of functions and a sequence of values. Returns the value or `NULL` in the case of an error; an exception will be raised if `NULL` is returned.

`Py_BuildValue()` does not always build a tuple. It builds a tuple
only if its format string contains two or more format units. If the format string is empty, it returns `None`; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

When memory buffers are passed as parameters to supply data to build objects, as for the `s` and `s#` formats, the required data is copied. Buffers provided by the caller are never referenced by the objects created by `Py_BuildValue()`. In other words, if your code invokes `malloc()` and passes the allocated memory to `Py_BuildValue()`, your code is responsible for calling `free()` for that memory once `Py_BuildValue()` returns.

In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that the format unit will return; and the entry in [square] brackets is the type of the C value(s) to be passed.

The characters space, tab, colon and comma are ignored in format strings (but not within format units such as `s#`). This can be used to make long format strings a tad more readable.

`s` (string) [char *]
Convert a null-terminated C string to a Python object. If the C string pointer is `NULL`, `None` is used.

`s#` (string) [char *, int]
Convert a C string and its length to a Python object. If the C string pointer is `NULL`, the length is ignored and `None` is returned.

`z` (string or `None`) [char *]
Same as `s`. 
z# (string or None) [char *, int]
   Same as s#.

u (Unicode string) [Py_UNICODE *]
   Convert a null-terminated buffer of Unicode (UCS-2 or UCS-4) data to a Python Unicode object. If the Unicode buffer pointer is NULL, None is returned.

u# (Unicode string) [Py_UNICODE *, int]
   Convert a Unicode (UCS-2 or UCS-4) data buffer and its length to a Python Unicode object. If the Unicode buffer pointer is NULL, the length is ignored and None is returned.

i (integer) [int]
   Convert a plain C int to a Python integer object.

b (integer) [char]
   Convert a plain C char to a Python integer object.

h (integer) [short int]
   Convert a plain C short int to a Python integer object.

l (integer) [long int]
   Convert a C long int to a Python integer object.

B (integer) [unsigned char]
   Convert a C unsigned char to a Python integer object.

H (integer) [unsigned short int]
   Convert a C unsigned short int to a Python integer object.

I (integer/long) [unsigned int]
   Convert a C unsigned int to a Python integer object or a Python long integer object, if it is larger than sys.maxint.
Convert a C unsigned long to a Python integer object or a Python long integer object, if it is larger than \texttt{sys.maxint}.

\texttt{L} (long) [PY\_LONG\_LONG]

Convert a C long \texttt{long} to a Python long integer object. Only available on platforms that support \texttt{long long}.

\texttt{K} (long) [unsigned PY\_LONG\_LONG]

Convert a C unsigned long \texttt{long} to a Python long integer \texttt{long} object. Only available on platforms that support unsigned \texttt{long long}.

\texttt{n} (int) [Py\_ssize\_t]

Convert a C Py\_ssize\_t to a Python integer or long integer.

\textit{New in version 2.5.}

\texttt{c} (string of length 1) [char]

Convert a C int representing a character to a Python string of length 1.

\texttt{d} (float) [double]

Convert a C \texttt{double} to a Python floating point number.

\texttt{f} (float) [float]

Same as \texttt{d}.

\texttt{D} (complex) [Py\_complex \*]

Convert a C Py\_complex structure to a Python complex number.

\texttt{o} (object) [PyObject \*]

Pass a Python object untouched (except for its reference count, which is incremented by one). If the object passed in is a \texttt{NULL} pointer, it is assumed that this was caused because the call producing the argument found an error and set an
exception. Therefore, `Py_BuildValue()` will return `NULL` but won't raise an exception. If no exception has been raised yet, `SystemError` is set.

S (object) [PyObject *]
Same as `O`.

N (object) [PyObject *]
Same as `O`, except it doesn't increment the reference count on the object. Useful when the object is created by a call to an object constructor in the argument list.

O\& (object) [converter, anything]
Convert `anything` to a Python object through a `converter` function. The function is called with `anything` (which should be compatible with `void *`) as its argument and should return a "new" Python object, or `NULL` if an error occurred.

(items) (tuple) [matching-items]
Convert a sequence of C values to a Python tuple with the same number of items.

[items] (list) [matching-items]
Convert a sequence of C values to a Python list with the same number of items.

{items} (dictionary) [matching-items]
Convert a sequence of C values to a Python dictionary. Each pair of consecutive C values adds one item to the dictionary, serving as key and value, respectively.

If there is an error in the format string, the `SystemError` exception is set and `NULL` returned.

`PyObject* Py_VaBuildValue(const char *format, va_list vars)`
Identical to `Py_BuildValue()`, except that it accepts a `va_list`
rather than a variable number of arguments.
String conversion and formatting

Functions for number conversion and formatted string output.

```c
int PyOS_snprintf(char *str, size_t size, const char *format, ...)  
Output not more than size bytes to str according to the format string format and the extra arguments. See the Unix man page snprintf(2).
```

```c
int PyOS_vsnprintf(char *str, size_t size, const char *format, va_list va)  
Output not more than size bytes to str according to the format string format and the variable argument list va. Unix man page vsnprintf(2).
```

PyOS_snprintf() and PyOS_vsnprintf() wrap the Standard C library functions snprintf() and vsnprintf(). Their purpose is to guarantee consistent behavior in corner cases, which the Standard C functions do not.

The wrappers ensure that str[\*size-1] is always '\0' upon return. They never write more than size bytes (including the trailing '\0' into str. Both functions require that str != NULL, size > 0 and format != NULL.

If the platform doesn’t have vsnprintf() and the buffer size needed to avoid truncation exceeds size by more than 512 bytes, Python aborts with a Py_FatalError.

The return value (rv) for these functions should be interpreted as follows:

- When 0 <= rv < size, the output conversion was successful and rv characters were written to str (excluding the trailing '\0')
byte at \texttt{str\texttt{\[*rv\]}}.

- When \texttt{rv >= size}, the output conversion was truncated and a buffer with \texttt{rv + 1} bytes would have been needed to succeed. \texttt{str\texttt{\[*size-1\]}} is '\texttt{\0}' in this case.
- When \texttt{rv < 0}, "something bad happened." \texttt{str\texttt{\[*size-1\]}} is '\texttt{\0}' in this case too, but the rest of \texttt{str} is undefined. The exact cause of the error depends on the underlying platform.

The following functions provide locale-independent string to number conversions.

\texttt{double PyOS\_string\_to\_double(const char \*s, char **endptr, PyObject \*overflow\_exception)}

Convert a string \texttt{s} to a \texttt{double}, raising a Python exception on failure. The set of accepted strings corresponds to the set of strings accepted by Python's \texttt{float()} constructor, except that \texttt{s} must not have leading or trailing whitespace. The conversion is independent of the current locale.

If \texttt{endptr} is \texttt{NULL}, convert the whole string. Raise ValueError and return \texttt{-1.0} if the string is not a valid representation of a floating-point number.

If \texttt{endptr} is not \texttt{NULL}, convert as much of the string as possible and set \texttt{*endptr} to point to the first unconverted character. If no initial segment of the string is the valid representation of a floating-point number, set \texttt{*endptr} to point to the beginning of the string, raise ValueError, and return \texttt{-1.0}.

If \texttt{s} represents a value that is too large to store in a float (for example, "1e500" is such a string on many platforms) then if \texttt{overflow\_exception} is \texttt{NULL} return \texttt{Py\_HUGE\_VAL} (with an appropriate sign) and don't set any exception. Otherwise, \texttt{overflow\_exception} must point to a Python exception object;
raise that exception and return -1.0. In both cases, set *endptr to point to the first character after the converted value.

If any other error occurs during the conversion (for example an out-of-memory error), set the appropriate Python exception and return -1.0.

*New in version 2.7.*

double **PyOS_ascii_strtod**(const char *nptr, char **endptr)

Convert a string to a double. This function behaves like the Standard C function **strtod()** does in the C locale. It does this without changing the current locale, since that would not be thread-safe.

**PyOS_ascii_strtod()** should typically be used for reading configuration files or other non-user input that should be locale independent.

See the Unix man page **strtod(2)** for details.

*New in version 2.4.*

**Deprecation: since version 2.7:** Use **PyOS_string_to_double()** instead.

char* **PyOS_ascii_formatd**(char *buffer, size_t buf_len, const char *format, double d)

Convert a double to a string using the '.' as the decimal separator. **format** is a printf()-style format string specifying the number format. Allowed conversion characters are 'e', 'E', 'f', 'F', 'g' and 'G'.

The return value is a pointer to buffer with the converted string or
NULL if the conversion failed.

New in version 2.4.

Deprecated since version 2.7: This function is removed in Python 2.7 and 3.1. Use PyOS_double_to_string() instead.

char* PyOS_double_to_string(double val, char format_code, int precision, int flags, int *ptype)

Convert a double val to a string using supplied format_code, precision, and flags.

format_code must be one of 'e', 'E', 'f', 'F', 'g', 'G' or 'r'. For 'r', the supplied precision must be 0 and is ignored. The 'r' format code specifies the standard repr() format.

flags can be zero or more of the values Py_DTSF_SIGN, Py_DTSF_ADD_DOT_0, or Py_DTSF_ALT, or-ed together:

- **Py_DTSF_SIGN** means to always precede the returned string with a sign character, even if val is non-negative.
- **Py_DTSF_ADD_DOT_0** means to ensure that the returned string will not look like an integer.
- **Py_DTSF_ALT** means to apply “alternate” formatting rules. See the documentation for the PyOS_snprintf() '#' specifier for details.

If ptype is non-NULL, then the value it points to will be set to one of Py_DTST_FINITE, Py_DTST_INFINITY, or Py_DTST_NAN, signifying that val is a finite number, an infinite number, or not a number, respectively.

The return value is a pointer to buffer with the converted string or NULL if the conversion failed. The caller is responsible for freeing the returned string by calling PyMem_Free().
New in version 2.7.

double PyOS_ascii_atof(const char *nptr)
    Convert a string to a double in a locale-independent way.

See the Unix man page atof(2) for details.

New in version 2.4.

**Deprecated since version 3.1:** Use PyOS_string_to_double() instead.

char* PyOS_stricmp(char *s1, char *s2)
    Case insensitive comparison of strings. The function works almost identically to strcmp() except that it ignores the case.

New in version 2.6.

char* PyOS_strnicmp(char *s1, char *s2, Py_ssize_t size)
    Case insensitive comparison of strings. The function works almost identically to strncmp() except that it ignores the case.

New in version 2.6.
Reflection

PyObject* PyEval_GetBuiltin()  
*Return value: Borrowed reference.*  
Return a dictionary of the builtins in the current execution frame,  
or the interpreter of the thread state if no frame is currently executing.

PyObject* PyEval_GetLocals()  
*Return value: Borrowed reference.*  
Return a dictionary of the local variables in the current execution frame, or *NULL* if no frame is currently executing.

PyObject* PyEval_GetGlobals()  
*Return value: Borrowed reference.*  
Return a dictionary of the global variables in the current execution frame, or *NULL* if no frame is currently executing.

PyFrameObject* PyEval_GetFrame()  
*Return value: Borrowed reference.*  
Return the current thread state's frame, which is *NULL* if no frame is currently executing.

int PyFrame_GetLineNumber(PyFrameObject *frame)  
Return the line number that *frame* is currently executing.

int PyEval_GetRestricted()  
If there is a current frame and it is executing in restricted mode,  
return true, otherwise false.

const char* PyEval_GetFuncName(PyObject *func)  
Return the name of *func* if it is a function, class or instance object, else the name of *funcs* type.

const char* PyEval_GetFuncDesc(PyObject *func)
Return a description string, depending on the type of `func`. Return values include “()” for functions and methods, ” constructor”, ” instance”, and ” object”. Concatenated with the result of `PyEval_GetFuncName()`, the result will be a description of `func`.
Codec registry and support functions

int PyCodec_Register(PyObject *search_function)

Register a new codec search function.

As side effect, this tries to load the encodings package, if not yet done, to make sure that it is always first in the list of search functions.

int PyCodec_KnownEncoding(const char *encoding)

Return 1 or 0 depending on whether there is a registered codec for the given encoding.

PyObject* PyCodec_Encode(PyObject *object, const char *encoding, const char *errors)

Generic codec based encoding API.

object is passed through the encoder function found for the given encoding using the error handling method defined by errors. errors may be NULL to use the default method defined for the codec. Raises a LookupError if no encoder can be found.

PyObject* PyCodec_Decode(PyObject *object, const char *encoding, const char *errors)

Generic codec based decoding API.

object is passed through the decoder function found for the given encoding using the error handling method defined by errors. errors may be NULL to use the default method defined for the codec. Raises a LookupError if no encoder can be found.
Codec lookup API

In the following functions, the `encoding` string is looked up converted to all lower-case characters, which makes encodings looked up through this mechanism effectively case-insensitive. If no codec is found, a `KeyError` is set and `NULL` returned.

```c
PyObject* PyCodec_Encoder(const char *encoding)
    Get an encoder function for the given `encoding`.

PyObject* PyCodec_Decoder(const char *encoding)
    Get a decoder function for the given `encoding`.

PyObject* PyCodec_IncrementalEncoder(const char *encoding, const char *errors)
    Get an IncrementalEncoder object for the given `encoding`.

PyObject* PyCodec_IncrementalDecoder(const char *encoding, const char *errors)
    Get an IncrementalDecoder object for the given `encoding`.

PyObject* PyCodec_StreamReader(const char *encoding, PyObject *stream, const char *errors)
    Get a StreamReader factory function for the given `encoding`.

PyObject* PyCodec_StreamWriter(const char *encoding, PyObject *stream, const char *errors)
    Get a StreamWriter factory function for the given `encoding`.
```
Registry API for Unicode encoding error handlers

int PyCodec_RegisterError(const char *name, PyObject *error)

Register the error handling callback function error under the given name. This callback function will be called by a codec when it encounters unencodable characters/undecodable bytes and name is specified as the error parameter in the call to the encode/decode function.

The callback gets a single argument, an instance of UnicodeEncodeError, UnicodeDecodeError or UnicodeTranslateError that holds information about the problematic sequence of characters or bytes and their offset in the original string (see Unicode Exception Objects for functions to extract this information). The callback must either raise the given exception, or return a two-item tuple containing the replacement for the problematic sequence, and an integer giving the offset in the original string at which encoding/decoding should be resumed.

Return 0 on success, -1 on error.

PyObject* PyCodec_LookupError(const char *name)

Lookup the error handling callback function registered under name. As a special case NULL can be passed, in which case the error handling callback for “strict” will be returned.

PyObject* PyCodec STRICTErrors(PyObject *exc)

Raise exc as an exception.

PyObject* PyCodec IgnoreErrors(PyObject *exc)

Ignore the unicode error, skipping the faulty input.
PyObject* PyCodec_ReplaceErrors(PyObject *exc)
Replace the unicode encode error with \? or \u+FFFF.

PyObject* PyCodec_XMLCharRefReplaceErrors(PyObject *exc)
Replace the unicode encode error with XML character references.

PyObject* PyCodec_BackslashReplaceErrors(PyObject *exc)
Replace the unicode encode error with backslash escapes (\x, \u and \U).
Abstract Objects Layer

The functions in this chapter interact with Python objects regardless of their type, or with wide classes of object types (e.g. all numerical types, or all sequence types). When used on object types for which they do not apply, they will raise a Python exception.

It is not possible to use these functions on objects that are not properly initialized, such as a list object that has been created by `PyList_New()`, but whose items have not been set to some non-NULL value yet.

- Object Protocol
- Number Protocol
- Sequence Protocol
- Mapping Protocol
- Iterator Protocol
- Old Buffer Protocol
Object Protocol

int PyObject_Print(PyObject *o, FILE *fp, int flags)
Print an object o, on file fp. Returns -1 on error. The flags argument is used to enable certain printing options. The only option currently supported is Py_PRINT_RAW; if given, the str() of the object is written instead of the repr().

int PyObject_HasAttr(PyObject *o, PyObject *attr_name)
Returns 1 if o has the attribute attr_name, and 0 otherwise. This is equivalent to the Python expression hasattr(o, attr_name). This function always succeeds.

int PyObject_HasAttrString(PyObject *o, const char *attr_name)
Returns 1 if o has the attribute attr_name, and 0 otherwise. This is equivalent to the Python expression hasattr(o, attr_name). This function always succeeds.

PyObject* PyObject_GetAttr(PyObject *o, PyObject *attr_name)
Retrieve an attribute named attr_name from object o. Returns the attribute value on success, or NULL on failure. This is the equivalent of the Python expression o.attr_name.

PyObject* PyObject_GetAttrString(PyObject *o, const char *attr_name)
Retrieve an attribute named attr_name from object o. Returns the attribute value on success, or NULL on failure. This is the equivalent of the Python expression o.attr_name.

PyObject* PyObject_GenericGetAttr(PyObject *o, PyObject *name)
Generic attribute getter function that is meant to be put into a type object's `tp_getattro` slot. It looks for a descriptor in the dictionary of classes in the object's MRO as well as an attribute in the object's `__dict__` (if present). As outlined in Implementing Descriptors, data descriptors take preference over instance attributes, while non-data descriptors don't. Otherwise, an `AttributeError` is raised.

```c
int PyObject_SetAttr(PyObject *o, PyObject *attr_name, PyObject *v)
```

Set the value of the attribute named `attr_name`, for object `o`, to the value `v`. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o.attr_name = v`.

If `v` is `NULL`, the attribute is deleted, however this feature is deprecated in favour of using `PyObject_DelAttr()`.

```c
int PyObject_SetAttrString(PyObject *o, const char *attr_name, PyObject *v)
```

Set the value of the attribute named `attr_name`, for object `o`, to the value `v`. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o.attr_name = v`.

If `v` is `NULL`, the attribute is deleted, however this feature is deprecated in favour of using `PyObject_DelAttrString()`.

```c
int PyObject_GenericSetAttr(PyObject *o, PyObject *name, PyObject *value)
```

Generic attribute setter and deleter function that is meant to be put into a type object's `tp_setattro` slot. It looks for a data descriptor in the dictionary of classes in the object's MRO, and if found it takes preference over setting or deleting the attribute in
the instance dictionary. Otherwise, the attribute is set or deleted in the object's `__dict__` (if present). On success, `0` is returned, otherwise an `AttributeError` is raised and `-1` is returned.

```c
int PyObject_DelAttr(PyObject *o, PyObject *attr_name)
    Delete attribute named `attr_name`, for object `o`. Returns `-1` on failure. This is the equivalent of the Python statement `del o.attr_name`.
```

```c
int PyObject_DelAttrString(PyObject *o, const char *attr_name)
    Delete attribute named `attr_name`, for object `o`. Returns `-1` on failure. This is the equivalent of the Python statement `del o.attr_name`.
```

```c
PyObject* PyObject_RichCompare(PyObject *o1, PyObject *o2, int opid)
    Compare the values of `o1` and `o2` using the operation specified by `opid`, which must be one of `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, or `Py_GE`, corresponding to `<`, `<=`, `==`, `!=`, `>`, or `>=` respectively. This is the equivalent of the Python expression `o1 op o2`, where `op` is the operator corresponding to `opid`. Returns the value of the comparison on success, or `NULL` on failure.
```

```c
int PyObject_RichCompareBool(PyObject *o1, PyObject *o2, int opid)
    Compare the values of `o1` and `o2` using the operation specified by `opid`, which must be one of `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, or `Py_GE`, corresponding to `<`, `<=`, `==`, `!=`, `>`, or `>=` respectively. Returns `-1` on error, `0` if the result is false, `1` otherwise. This is the equivalent of the Python expression `o1 op o2`, where `op` is the operator corresponding to `opid`.
```
**Note:** If \( o1 \) and \( o2 \) are the same object, `PyObject_RichCompareBool()` will always return 1 for `Py_EQ` and 0 for `Py_NE`.

```c
int PyObject_Cmp(PyObject *o1, PyObject *o2, int *result)
```

Compare the values of \( o1 \) and \( o2 \) using a routine provided by \( o1 \), if one exists, otherwise with a routine provided by \( o2 \). The result of the comparison is returned in \( result \). Returns -1 on failure. This is the equivalent of the Python statement `result = cmp(o1, o2)`.

```c
int PyObject_Compare(PyObject *o1, PyObject *o2)
```

Compare the values of \( o1 \) and \( o2 \) using a routine provided by \( o1 \), if one exists, otherwise with a routine provided by \( o2 \). Returns the result of the comparison on success. On error, the value returned is undefined; use `PyErr_Occurred()` to detect an error. This is equivalent to the Python expression `cmp(o1, o2)`.

```c
PyObject* PyObject_Repr(PyObject *o)
```

*Return value: New reference.*

Compute a string representation of object \( o \). Returns the string representation on success, `NULL` on failure. This is the equivalent of the Python expression `repr(o)`. Called by the `repr()` built-in function and by reverse quotes.

```c
PyObject* PyObject_Str(PyObject *o)
```

*Return value: New reference.*

Compute a string representation of object \( o \). Returns the string representation on success, `NULL` on failure. This is the equivalent of the Python expression `str(o)`. Called by the `str()` built-in function and by the `print` statement.

```c
PyObject* PyObjectBytes(PyObject *o)
```

Compute a bytes representation of object \( o \). In 2.x, this is just an
alias for `PyObject_Str()`.

**PyObject* `PyObject_Unicode(PyObject *o)`**

*Return value: New reference.*

Compute a Unicode string representation of object `o`. Returns the Unicode string representation on success, `NULL` on failure. This is the equivalent of the Python expression `unicode(o)`. Called by the `unicode()` built-in function.

```c
int PyObject_IsInstance(PyObject *inst, PyObject *cls)
```

Returns `1` if `inst` is an instance of the class `cls` or a subclass of `cls`, or `0` if not. On error, returns `-1` and sets an exception. If `cls` is a type object rather than a class object, `PyObject_IsInstance()` returns `1` if `inst` is of type `cls`. If `cls` is a tuple, the check will be done against every entry in `cls`. The result will be `1` when at least one of the checks returns `1`, otherwise it will be `0`. If `inst` is not a class instance and `cls` is neither a type object, nor a class object, nor a tuple, `inst` must have a `__class__` attribute — the class relationship of the value of that attribute with `cls` will be used to determine the result of this function.

*New in version 2.1.*

*Changed in version 2.2:* Support for a tuple as the second argument added.

Subclass determination is done in a fairly straightforward way, but includes a wrinkle that implementors of extensions to the class system may want to be aware of. If `A` and `B` are class objects, `B` is a subclass of `A` if it inherits from `A` either directly or indirectly. If either is not a class object, a more general mechanism is used to determine the class relationship of the two objects. When testing if `B` is a subclass of `A`, if `A` is `B`, `PyObject_IsSubclass()` returns true. If `A` and `B` are different objects, `B`'s `__bases__` attribute is searched in a
depth-first fashion for A — the presence of the __bases__ attribute is considered sufficient for this determination.

```c
int PyObject_IsSubclass(PyObject *derived, PyObject *cls)

Returns 1 if the class derived is identical to or derived from the class cls, otherwise returns 0. In case of an error, returns -1. If cls is a tuple, the check will be done against every entry in cls. The result will be 1 when at least one of the checks returns 1, otherwise it will be 0. If either derived or cls is not an actual class object (or tuple), this function uses the generic algorithm described above.

New in version 2.1.

Changed in version 2.3: Older versions of Python did not support a tuple as the second argument.
```

```c
int PyCallable_Check(PyObject *o)

Determine if the object o is callable. Return 1 if the object is callable and 0 otherwise. This function always succeeds.

PyObject* PyObject_Call(PyObject *callable_object, PyObject *args, PyObject *kw)


Call a callable Python object callable_object, with arguments given by the tuple args, and named arguments given by the dictionary kw. If no named arguments are needed, kw may be NULL. args must not be NULL, use an empty tuple if no arguments are needed. Returns the result of the call on success, or NULL on failure. This is the equivalent of the Python expression apply(callable_object, args, kw) or callable_object(*args, **kw).

New in version 2.2.
PyObject* PyObject_CallObject(PyObject *callable_object, PyObject *args)

*Return value: New reference.*

Call a callable Python object *callable_object*, with arguments given by the tuple *args*. If no arguments are needed, then *args* may be *NULL*. Returns the result of the call on success, or *NULL* on failure. This is the equivalent of the Python expression `apply(callable_object, args)` or `callable_object(*args)`.

PyObject* PyObject_CallFunction(PyObject *callable, char *format, ...)

*Return value: New reference.*

Call a callable Python object *callable*, with a variable number of C arguments. The C arguments are described using a `Py_BuildValue()` style format string. The format may be *NULL*, indicating that no arguments are provided. Returns the result of the call on success, or *NULL* on failure. This is the equivalent of the Python expression `apply(callable, args)` or `callable(*args)`. Note that if you only pass `PyObject * args`, `PyObject_CallFunctionObjArgs()` is a faster alternative.

PyObject* PyObject_CallMethod(PyObject *o, char *method, char *format, ...)

*Return value: New reference.*

Call the method named *method* of object *o* with a variable number of C arguments. The C arguments are described by a `Py_BuildValue()` format string that should produce a tuple. The format may be *NULL*, indicating that no arguments are provided. Returns the result of the call on success, or *NULL* on failure. This is the equivalent of the Python expression `o.method(args)`. Note that if you only pass `PyObject * args`, `PyObject_CallMethodObjArgs()` is a faster alternative.

PyObject* PyObject_CallFunctionObjArgs(PyObject *callable, ...)
Call a callable Python object callable, with a variable number of PyObject* arguments. The arguments are provided as a variable number of parameters followed by NULL. Returns the result of the call on success, or NULL on failure.

New in version 2.2.

PyObject* PyObject_CallMethodObjArgs(PyObject *o, PyObject *name, ..., NULL)
Calls a method of the object o, where the name of the method is given as a Python string object in name. It is called with a variable number of PyObject* arguments. The arguments are provided as a variable number of parameters followed by NULL. Returns the result of the variable number of parameters followed by NULL.

New in version 2.2.

long PyObject_Hash(PyObject *o)
Compute and return the hash value of an object o. On failure, return -1. This is the equivalent of the Python expression hash(o).

long PyObject_HashNotImplemented(PyObject *o)
Set a TypeError indicating that type(o) is not hashable and return -1. This function receives special treatment when stored in a tp_hash slot, allowing a type to explicitly indicate to the interpreter that it is not hashable.

New in version 2.6.

int PyObject_IsTrue(PyObject *o)
Returns 1 if the object o is considered to be true, and 0
otherwise. This is equivalent to the Python expression `not not o`. On failure, return `-1`.

```c
int PyObject_Not(PyObject *o)
```

Returns 0 if the object `o` is considered to be true, and 1 otherwise. This is equivalent to the Python expression `not o`. On failure, return `-1`.

```c
PyObject* PyObject_Type(PyObject *o)
```

*Return value: New reference.*

When `o` is non-NULL, returns a type object corresponding to the object type of object `o`. On failure, raises `SystemError` and returns `NULL`. This is equivalent to the Python expression `type(o)`. This function increments the reference count of the return value. There's really no reason to use this function instead of the common expression `o->ob_type`, which returns a pointer of type `PyTypeObject*`, except when the incremented reference count is needed.

```c
int PyObject_TypeCheck(PyObject *o, PyTypeObject *type)
```

Return true if the object `o` is of type `type` or a subtype of `type`. Both parameters must be non-NULL.

*New in version 2.2.*

```c
Py_ssize_t PyObject_Length(PyObject *o)
Py_ssize_t PyObject_Size(PyObject *o)
```

Return the length of object `o`. If the object `o` provides either the sequence and mapping protocols, the sequence length is returned. On error, `-1` is returned. This is the equivalent to the Python expression `len(o)`.

*Changed in version 2.5:* These functions returned an `int` type. This might require changes in your code for properly supporting
64-bit systems.

**PyObject** PyObject**_GetItem**(PyObject *o, PyObject *key)

*Return value: New reference.*

Return element of o corresponding to the object key or NULL on failure. This is the equivalent of the Python expression o[key].

**int** PyObject**_SetItem**(PyObject *o, PyObject *key, PyObject *v)

Map the object key to the value v. Raise an exception and return -1 on failure; return 0 on success. This is the equivalent of the Python statement o[key] = v.

**int** PyObject**_DelItem**(PyObject *o, PyObject *key)

Delete the mapping for key from o. Returns -1 on failure. This is the equivalent of the Python statement del o[key].

**int** PyObject**_AsFileDescriptor**(PyObject *o)

Derives a file descriptor from a Python object. If the object is an integer or long integer, its value is returned. If not, the object’s fileno() method is called if it exists; the method must return an integer or long integer, which is returned as the file descriptor value. Returns -1 on failure.

**PyObject*** PyObject**_Dir**(PyObject *o)

*Return value: New reference.*

This is equivalent to the Python expression dir(o), returning a (possibly empty) list of strings appropriate for the object argument, or NULL if there was an error. If the argument is NULL, this is like the Python dir(), returning the names of the current locals; in this case, if no execution frame is active then NULL is returned but PyErr_occurred() will return false.

**PyObject*** PyObject**_GetIter**(PyObject *o)

*Return value: New reference.*
This is equivalent to the Python expression `iter(o)`. It returns a new iterator for the object argument, or the object itself if the object is already an iterator. Raises `TypeError` and returns `NULL` if the object cannot be iterated.
Number Protocol

```c
int PyNumber_Check(PyObject *o)

    Returns 1 if the object o provides numeric protocols, and false otherwise. This function always succeeds.
```

```c
PyObject* PyNumber_Add(PyObject *o1, PyObject *o2)

    Returns the result of adding o1 and o2, or NULL on failure. This is the equivalent of the Python expression o1 + o2.
```

```c
PyObject* PyNumber_Subtract(PyObject *o1, PyObject *o2)

    Returns the result of subtracting o2 from o1, or NULL on failure. This is the equivalent of the Python expression o1 - o2.
```

```c
PyObject* PyNumber_Multiply(PyObject *o1, PyObject *o2)

    Returns the result of multiplying o1 and o2, or NULL on failure. This is the equivalent of the Python expression o1 * o2.
```

```c
PyObject* PyNumber_Divide(PyObject *o1, PyObject *o2)

    Returns the result of dividing o1 by o2, or NULL on failure. This is the equivalent of the Python expression o1 / o2.
```

```c
PyObject* PyNumber_FloorDivide(PyObject *o1, PyObject *o2)

    Return the floor of o1 divided by o2, or NULL on failure. This is equivalent to the “classic” division of integers.

    New in version 2.2.
```

```c
PyObject* PyNumber_TrueDivide(PyObject *o1, PyObject *o2)
```
Return a reasonable approximation for the mathematical value of $o1$ divided by $o2$, or NULL on failure. The return value is “approximate” because binary floating point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers.

New in version 2.2.

PyObject* PyNumber_Remainder(PyObject *o1, PyObject *o2)
Returns the remainder of dividing $o1$ by $o2$, or NULL on failure. This is the equivalent of the Python expression $o1 \% o2$.

PyObject* PyNumber_Divmod(PyObject *o1, PyObject *o2)
See the built-in function divmod(). Returns NULL on failure. This is the equivalent of the Python expression divmod(o1, o2).

PyObject* PyNumber_Power(PyObject *o1, PyObject *o2, PyObject *o3)
See the built-in function pow(). Returns NULL on failure. This is the equivalent of the Python expression pow(o1, o2, o3), where o3 is optional. If o3 is to be ignored, pass Py_None in its place (passing NULL for o3 would cause an illegal memory access).

PyObject* PyNumber_Negative(PyObject *o)
Returns the negation of o on success, or NULL on failure. This is the equivalent of the Python expression -o.

PyObject* PyNumber_Positive(PyObject *o)
Returns \( o \) on success, or \( NULL \) on failure. This is the equivalent of the Python expression \(+o\).

\section*{PyObject* PyNumber_Absolute(PyObject *o)}

\textit{Return value: New reference.}

Returns the absolute value of \( o \), or \( NULL \) on failure. This is the equivalent of the Python expression \( \text{abs}(o) \).

\section*{PyObject* PyNumber_Invert(PyObject *o)}

\textit{Return value: New reference.}

Returns the bitwise negation of \( o \) on success, or \( NULL \) on failure. This is the equivalent of the Python expression \( \sim o \).

\section*{PyObject* PyNumber_Lshift(PyObject *o1, PyObject *o2)}

\textit{Return value: New reference.}

Returns the result of left shifting \( o1 \) by \( o2 \) on success, or \( NULL \) on failure. This is the equivalent of the Python expression \( o1 \ll o2 \).

\section*{PyObject* PyNumber_Rshift(PyObject *o1, PyObject *o2)}

\textit{Return value: New reference.}

Returns the result of right shifting \( o1 \) by \( o2 \) on success, or \( NULL \) on failure. This is the equivalent of the Python expression \( o1 \gg o2 \).

\section*{PyObject* PyNumber_And(PyObject *o1, PyObject *o2)}

\textit{Return value: New reference.}

Returns the “bitwise and” of \( o1 \) and \( o2 \) on success and \( NULL \) on failure. This is the equivalent of the Python expression \( o1 \& o2 \).

\section*{PyObject* PyNumber_Xor(PyObject *o1, PyObject *o2)}

\textit{Return value: New reference.}

Returns the “bitwise exclusive or” of \( o1 \) by \( o2 \) on success, or \( NULL \) on failure. This is the equivalent of the Python expression \( o1 \oplus o2 \).
PyObject* PyNumber_Or(PyObject *o1, PyObject *o2)

*Return value: New reference.*

Returns the “bitwise or” of o1 and o2 on success, or NULL on failure. This is the equivalent of the Python expression `o1 | o2`.

PyObject* PyNumber_InPlaceAdd(PyObject *o1, PyObject *o2)

*Return value: New reference.*

Returns the result of adding o1 and o2, or NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the Python statement `o1 += o2`.

PyObject* PyNumber_InPlaceSubtract(PyObject *o1, PyObject *o2)

*Return value: New reference.*

Returns the result of subtracting o2 from o1, or NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the Python statement `o1 -= o2`.

PyObject* PyNumber_InPlaceMultiply(PyObject *o1, PyObject *o2)

*Return value: New reference.*

Returns the result of multiplying o1 and o2, or NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the Python statement `o1 *= o2`.

PyObject* PyNumber_InPlaceDivide(PyObject *o1, PyObject *o2)

*Return value: New reference.*

Returns the result of dividing o1 by o2, or NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the Python statement `o1 /= o2`.

PyObject* PyNumber_InPlaceFloorDivide(PyObject *o1, PyObject *o2)
Returns the mathematical floor of dividing o1 by o2, or NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python statement o1 /= o2.

New in version 2.2.

PyObject* PyNumber_InPlaceTrueDivide(PyObject *o1, PyObject *o2)

Return a reasonable approximation for the mathematical value of o1 divided by o2, or NULL on failure. The return value is “approximate” because binary floating point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers. The operation is done in-place when o1 supports it.

New in version 2.2.

PyObject* PyNumber_InPlaceRemainder(PyObject *o1, PyObject *o2)

Returns the remainder of dividing o1 by o2, or NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python statement o1 %= o2.

PyObject* PyNumber_InPlacePower(PyObject *o1, PyObject *o2, PyObject *o3)

See the built-in function pow(). Returns NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python statement o1 **= o2 when o3 is Py_None, or an in-place variant of pow(o1, o2, o3) otherwise. If o3 is to be ignored, pass Py_None in its place (passing NULL for
o3 would cause an illegal memory access).

**PyObject* PyNumber_InPlaceLshift(PyObject *o1, PyObject *o2)**

*Return value: New reference.*

Returns the result of left shifting o1 by o2 on success, or NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the Python statement o1 <<= o2.

**PyObject* PyNumber_InPlaceRshift(PyObject *o1, PyObject *o2)**

*Return value: New reference.*

Returns the result of right shifting o1 by o2 on success, or NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the Python statement o1 >>= o2.

**PyObject* PyNumber_InPlaceAnd(PyObject *o1, PyObject *o2)**

*Return value: New reference.*

Returns the “bitwise and” of o1 and o2 on success and NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the Python statement o1 &= o2.

**PyObject* PyNumber_InPlaceXor(PyObject *o1, PyObject *o2)**

*Return value: New reference.*

Returns the “bitwise exclusive or” of o1 by o2 on success, or NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the Python statement o1 ^= o2.

**PyObject* PyNumber_InPlaceOr(PyObject *o1, PyObject *o2)**

*Return value: New reference.*

Returns the “bitwise or” of o1 and o2 on success, or NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the Python statement o1 |= o2.

**int PyNumber_Coerce(PyObject **p1, PyObject **p2)**

This function takes the addresses of two variables of type
PyObject*. If the objects pointed to by *p1 and *p2 have the same type, increment their reference count and return 0 (success). If the objects can be converted to a common numeric type, replace *p1 and *p2 by their converted value (with ‘new’ reference counts), and return 0. If no conversion is possible, or if some other error occurs, return -1 (failure) and don’t increment the reference counts. The call PyNumber_Coerce(&o1, &o2) is equivalent to the Python statement o1, o2 = coerce(o1, o2).

int PyNumber_CoerceEx(PyObject **p1, PyObject **p2)
This function is similar to PyNumber_Coerce(), except that it returns 1 when the conversion is not possible and when no error is raised. Reference counts are still not increased in this case.

PyObject* PyNumber_Int(PyObject *o)
Returns the o converted to an integer object on success, or NULL on failure. If the argument is outside the integer range a long object will be returned instead. This is the equivalent of the Python expression int(o).

PyObject* PyNumber_Long(PyObject *o)
Returns the o converted to a long integer object on success, or NULL on failure. This is the equivalent of the Python expression long(o).

PyObject* PyNumber_Float(PyObject *o)
Returns the o converted to a float object on success, or NULL on failure. This is the equivalent of the Python expression float(o).

PyObject* PyNumber_Index(PyObject *o)
Returns the o converted to a Python int or long on success or
NULL with a TypeError exception raised on failure.

New in version 2.5.

PyObject* PyNumber_ToBase(PyObject *n, int base)
Returns the integer n converted to base as a string with a base marker of '0b', '0o', or '0x' if applicable. When base is not 2, 8, 10, or 16, the format is 'x#num' where x is the base. If n is not an int object, it is converted with PyNumber_Index() first.

New in version 2.6.

Py_ssize_t PyNumber_AsSsize_t(PyObject *o, PyObject *exc)
Returns o converted to a Py_ssize_t value if o can be interpreted as an integer. If o can be converted to a Python int or long but the attempt to convert to a Py_ssize_t value would raise an OverflowError, then the exc argument is the type of exception that will be raised (usually IndexError or OverflowError). If exc is NULL, then the exception is cleared and the value is clipped to PY_SSIZE_T_MIN for a negative integer or PY_SSIZE_T_MAX for a positive integer.

New in version 2.5.

int PyIndex_Check(PyObject *o)
Returns 1 if o is an index integer (has the nb_index slot of the tp_as_number structure filled in), and 0 otherwise.

New in version 2.5.
Reference Manual » Abstract Objects Layer »
Sequence Protocol

int PySequence_Check(PyObject *o)
    Return 1 if the object provides sequence protocol, and 0 otherwise. This function always succeeds.

Py_ssize_t PySequence_Size(PyObject *o)  
Py_ssize_t PySequence_Length(PyObject *o)
    Returns the number of objects in sequence o on success, and -1 on failure. For objects that do not provide sequence protocol, this is equivalent to the Python expression `len(o)`.

  Changed in version 2.5: These functions returned an int type. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PySequence_Concat(PyObject *o1, PyObject *o2)
    Return the concatenation of o1 and o2 on success, and NULL on failure. This is the equivalent of the Python expression `o1 + o2`.

PyObject* PySequence_Repeat(PyObject *o, Py_ssize_t count)
    Return the result of repeating sequence object o count times, or NULL on failure. This is the equivalent of the Python expression `o * count`.

  Changed in version 2.5: This function used an int type for count. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PySequence_InPlaceConcat(PyObject *o1, PyObject *o2)
Return the concatenation of \( o1 \) and \( o2 \) on success, and \textit{NULL} on failure. The operation is done \textit{in-place} when \( o1 \) supports it. This is the equivalent of the Python expression \( o1 += o2 \).

\textbf{PyObject* PySequence\_InPlaceRepeat(PyObject *o, Py\_ssize\_t count)}

\textit{Return value: New reference.}
Return the result of repeating sequence object \( o \) \textit{count} times, or \textit{NULL} on failure. The operation is done \textit{in-place} when \( o \) supports it. This is the equivalent of the Python expression \( o *= count \).

\textit{Changed in version 2.5:} This function used an \texttt{int} type for \textit{count}. This might require changes in your code for properly supporting 64-bit systems.

\textbf{PyObject* PySequence\_GetItem(PyObject *o, Py\_ssize\_t i)}

\textit{Return value: New reference.}
Return the \textit{i}th element of \( o \), or \textit{NULL} on failure. This is the equivalent of the Python expression \( o[i] \).

\textit{Changed in version 2.5:} This function used an \texttt{int} type for \textit{i}. This might require changes in your code for properly supporting 64-bit systems.

\textbf{PyObject* PySequence\_GetSlice(PyObject *o, Py\_ssize\_t i1, Py\_ssize\_t i2)}

\textit{Return value: New reference.}
Return the slice of sequence object \( o \) between \( i1 \) and \( i2 \), or \textit{NULL} on failure. This is the equivalent of the Python expression \( o[i1:i2] \).

\textit{Changed in version 2.5:} This function used an \texttt{int} type for \textit{i1} and \textit{i2}. This might require changes in your code for properly supporting 64-bit systems.
int PySequence_SetItem(PyObject *o, Py_ssize_t i, PyObject *v)
    Assign object v to the i\(^{th}\) element of o. Raise an exception and return -1 on failure; return 0 on success. This is the equivalent of the Python statement \(o[i] = v\). This function does not steal a reference to v.

    If v is NULL, the element is deleted, however this feature is deprecated in favour of using PySequence_DelItem().

    Changed in version 2.5: This function used an int type for i. This might require changes in your code for properly supporting 64-bit systems.

int PySequence_DelItem(PyObject *o, Py_ssize_t i)
    Delete the i\(^{th}\) element of object o. Returns -1 on failure. This is the equivalent of the Python statement del o[i].

    Changed in version 2.5: This function used an int type for i. This might require changes in your code for properly supporting 64-bit systems.

int PySequence_SetSlice(PyObject *o, Py_ssize_t i1, Py_ssize_t i2, PyObject *v)
    Assign the sequence object v to the slice in sequence object o from i1 to i2. Raise an exception and return -1 on failure; return 0 on success. This is the equivalent of the Python statement \(o[i1:i2] = v\).

    If v is NULL, the slice is deleted, however this feature is deprecated in favour of using PySequence_DelSlice().

    Changed in version 2.5: This function used an int type for i1 and i2. This might require changes in your code for properly supporting 64-bit systems.
int PySequence_DelSlice(PyObject *o, Py_ssize_t i1, Py_ssize_t i2)
    Delete the slice in sequence object o from i1 to i2. Returns -1 on failure. This is the equivalent of the Python statement del o[i1:i2].

    Changed in version 2.5: This function used an int type for i1 and i2. This might require changes in your code for properly supporting 64-bit systems.

Py_ssize_t PySequence_Count(PyObject *o, PyObject *value)
    Return the number of occurrences of value in o, that is, return the number of keys for which o[key] == value. On failure, return -1. This is equivalent to the Python expression o.count(value).

    Changed in version 2.5: This function returned an int type. This might require changes in your code for properly supporting 64-bit systems.

int PySequence_Contains(PyObject *o, PyObject *value)
    Determine if o contains value. If an item in o is equal to value, return 1, otherwise return 0. On error, return -1. This is equivalent to the Python expression value in o.

Py_ssize_t PySequence_Index(PyObject *o, PyObject *value)
    Return the first index i for which o[i] == value. On error, return -1. This is equivalent to the Python expression o.index(value).

    Changed in version 2.5: This function returned an int type. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PySequence_List(PyObject *o)
Return a list object with the same contents as the arbitrary sequence o. The returned list is guaranteed to be new.

**PyObject** PySequence_Tuple(PyObject *o)

*Return value: New reference.*

Return a tuple object with the same contents as the arbitrary sequence o or NULL on failure. If o is a tuple, a new reference will be returned, otherwise a tuple will be constructed with the appropriate contents. This is equivalent to the Python expression tuple(o).

**PyObject** PySequence_Fast(PyObject *o, const char *m)

*Return value: New reference.*

Return the sequence o as a list, unless it is already a tuple or list, in which case o is returned. Use PySequence_Fast_GET_ITEM() to access the members of the result. Returns NULL on failure. If the object is not a sequence, raises TypeError with m as the message text.

**PyObject** PySequence_Fast_GET_ITEM(PyObject *o, Py_ssize_t i)

*Return value: Borrowed reference.*

Return the i-th element of o, assuming that o was returned by PySequence_Fast(), o is not NULL, and that i is within bounds.

*Changed in version 2.5:* This function used an int type for i. This might require changes in your code for properly supporting 64-bit systems.

**PyObject** PySequence_Fast_ITEMS(PyObject *o)

Return the underlying array of PyObject pointers. Assumes that o was returned by PySequence_Fast() and o is not NULL.

Note, if a list gets resized, the reallocation may relocate the items array. So, only use the underlying array pointer in contexts where the sequence cannot change.
New in version 2.4.

PyObject* PySequence_ITEM(PyObject *o, Py_ssize_t i)

Return the \textit{i}th element of \textit{o} or \texttt{NULL} on failure. Macro form of \*PySequence_GetItem() but without checking that \*PySequence_Check() on \textit{o} is true and without adjustment for negative indices.

New in version 2.3.

\textit{Changed in version 2.5}: This function used an \texttt{int} type for \textit{i}. This might require changes in your code for properly supporting 64-bit systems.

Py_ssize_t PySequence_Fast_GET_SIZE(PyObject *o)

Returns the length of \textit{o}, assuming that \textit{o} was returned by \*PySequence_Fast() and that \textit{o} is not \texttt{NULL}. The size can also be gotten by calling \texttt{PySequence_Size()} on \textit{o}, but \texttt{PySequence_Fast_GET_SIZE()} is faster because it can assume \textit{o} is a list or tuple.
Mapping Protocol

```c
int PyMapping_Check(PyObject *o)
    Return 1 if the object provides mapping protocol, and 0 otherwise. This function always succeeds.
```

```c
Py_ssize_t PyMapping_Size(PyObject *o)
Py_ssize_t PyMapping_Length(PyObject *o)
    Returns the number of keys in object o on success, and -1 on failure. For objects that do not provide mapping protocol, this is equivalent to the Python expression `len(o)`.
```

*Changed in version 2.5:* These functions returned an `int` type. This might require changes in your code for properly supporting 64-bit systems.

```c
int PyMapping_DelItemString(PyObject *o, char *key)
    Remove the mapping for object key from the object o. Return -1 on failure. This is equivalent to the Python statement `del o[key]`.
```

```c
int PyMapping_DelItem(PyObject *o, PyObject *key)
    Remove the mapping for object key from the object o. Return -1 on failure. This is equivalent to the Python statement `del o[key]`.
```

```c
int PyMapping_HasKeyString(PyObject *o, char *key)
    On success, return 1 if the mapping object has the key key and 0 otherwise. This is equivalent to `o[key]`, returning `True` on success and `False` on an exception. This function always succeeds.
```

```c
int PyMapping_HasKey(PyObject *o, PyObject *key)
    Return 1 if the mapping object has the key key and 0 otherwise. This is equivalent to `o[key]`, returning `True` on success and `False` on an exception. This function always succeeds.
```
PyObject* PyMapping_Keys(PyObject *o)

*Return value: New reference.*
On success, return a list of the keys in object o. On failure, return *NULL*. This is equivalent to the Python expression `o.keys()`.

PyObject* PyMapping_Values(PyObject *o)

*Return value: New reference.*
On success, return a list of the values in object o. On failure, return *NULL*. This is equivalent to the Python expression `o.values()`.

PyObject* PyMapping_Items(PyObject *o)

*Return value: New reference.*
On success, return a list of the items in object o, where each item is a tuple containing a key-value pair. On failure, return *NULL*. This is equivalent to the Python expression `o.items()`.

PyObject* PyMapping_GetItemString(PyObject *o, char *key)

*Return value: New reference.*
Return element of o corresponding to the object key or *NULL* on failure. This is the equivalent of the Python expression `o[key]`.

int PyMapping_SetItemString(PyObject *o, char *key, PyObject *v)

Map the object key to the value v in object o. Returns -1 on failure. This is the equivalent of the Python statement `o[key] = v`. 
Iterator Protocol

New in version 2.2.

There are two functions specifically for working with iterators.

```c
int PyIter_Check(PyObject *o)
    Return true if the object o supports the iterator protocol.
```

This function can return a false positive in the case of old-style classes because those classes always define a `tp_iter_next` slot with logic that either invokes a `next()` method or raises a `TypeError`.

```c
PyObject* PyIter_Next(PyObject *o)
    Return the next value from the iteration o. The object must be an iterator (it is up to the caller to check this). If there are no remaining values, returns `NULL` with no exception set. If an error occurs while retrieving the item, returns `NULL` and passes along the exception.
```

To write a loop which iterates over an iterator, the C code should look something like this:

```c
PyObject *iterator = PyObject_GetIter(obj);
PyObject *item;

if (iterator == NULL) {
    /* propagate error */
}

while (item = PyIter_Next(iterator)) {
    /* do something with item */
    ...
    /* release reference when done */
    Py_DECREF(item);
```
Py_DECREF(iterator);

if (PyErr_Occurred()) {
    /* propagate error */
} else {
    /* continue doing useful work */
}
This section describes the legacy buffer protocol, which has been introduced in Python 1.6. It is still supported but deprecated in the Python 2.x series. Python 3 introduces a new buffer protocol which fixes weaknesses and shortcomings of the protocol, and has been backported to Python 2.6. See Buffers and Memoryview Objects for more information.

```c
int PyObject_AsCharBuffer(PyObject *obj, const char **buffer, Py_ssize_t *buffer_len)
```

Returns a pointer to a read-only memory location usable as character-based input. The `obj` argument must support the single-segment character buffer interface. On success, returns 0, sets `buffer` to the memory location and `buffer_len` to the buffer length. Returns -1 and sets a `TypeError` on error.

*New in version 1.6.*

*Changed in version 2.5:* This function used an `int *` type for `buffer_len`. This might require changes in your code for properly supporting 64-bit systems.

```c
int PyObject_AsReadBuffer(PyObject *obj, const void **buffer, Py_ssize_t *buffer_len)
```

Returns a pointer to a read-only memory location containing arbitrary data. The `obj` argument must support the single-segment readable buffer interface. On success, returns 0, sets `buffer` to the memory location and `buffer_len` to the buffer length. Returns -1 and sets a `TypeError` on error.

*New in version 1.6.*

*Changed in version 2.5:* This function used an `int *` type for
buffer_len. This might require changes in your code for properly supporting 64-bit systems.

int PyObject_CheckReadBuffer(PyObject *o)
Returns 1 if o supports the single-segment readable buffer interface. Otherwise returns 0.

New in version 2.2.

int PyObject_AsWriteBuffer(PyObject *obj, void **buffer, Py_ssize_t *buffer_len)
Returns a pointer to a writeable memory location. The obj argument must support the single-segment, character buffer interface. On success, returns 0, sets buffer to the memory location and buffer_len to the buffer length. Returns -1 and sets a TypeError on error.

New in version 1.6.

Changed in version 2.5: This function used an int * type for buffer_len. This might require changes in your code for properly supporting 64-bit systems.
Concrete Objects Layer

The functions in this chapter are specific to certain Python object types. Passing them an object of the wrong type is not a good idea; if you receive an object from a Python program and you are not sure that it has the right type, you must perform a type check first; for example, to check that an object is a dictionary, use `PyDict_Check()`. The chapter is structured like the “family tree” of Python object types.

**Warning:** While the functions described in this chapter carefully check the type of the objects which are passed in, many of them do not check for *NULL* being passed instead of a valid object. Allowing *NULL* to be passed in can cause memory access violations and immediate termination of the interpreter.
Fundamental Objects

This section describes Python type objects and the singleton object `None`.

- Type Objects
- The `None` Object
Numeric Objects

- Plain Integer Objects
- Boolean Objects
- Long Integer Objects
- Floating Point Objects
- Complex Number Objects
  - Complex Numbers as C Structures
  - Complex Numbers as Python Objects
Sequence Objects

Generic operations on sequence objects were discussed in the previous chapter; this section deals with the specific kinds of sequence objects that are intrinsic to the Python language.

- **Byte Array Objects**
  - Type check macros
  - Direct API functions
  - Macros
- **String/Bytes Objects**
- **Unicode Objects and Codecs**
  - Unicode Objects
    - Unicode Type
    - Unicode Character Properties
    - Plain Py_UNICODE
    - wchar_t Support
  - Built-in Codecs
    - Generic Codecs
    - UTF-8 Codecs
    - UTF-32 Codecs
    - UTF-16 Codecs
    - UTF-7 Codecs
    - Unicode-Escape Codecs
    - Raw-Unicode-Escape Codecs
    - Latin-1 Codecs
    - ASCII Codecs
    - Character Map Codecs
    - MBCS codecs for Windows
    - Methods & Slots
  - Methods and Slot Functions
- **Buffers and Memoryview Objects**
  - The new-style Py_buffer struct
- Buffer related functions
- MemoryView objects
- Old-style buffer objects

- Tuple Objects
- List Objects
Mapping Objects

- Dictionary Objects
Other Objects

- Class and Instance Objects
- Function Objects
- Method Objects
- File Objects
- Module Objects
- Iterator Objects
- Descriptor Objects
- Slice Objects
- Weak Reference Objects
- Capsules
- CObjects
- Cell Objects
- Generator Objects
- DateTime Objects
- Set Objects
- Code Objects
Type Objects

**PyTypeObject**
The C structure of the objects used to describe built-in types.

**PyObject* PyType_Type**
This is the type object for type objects; it is the same object as `type` and `types.TypeType` in the Python layer.

```c
int PyType_Check(PyObject *o)
```
Return true if the object `o` is a type object, including instances of types derived from the standard type object. Return false in all other cases.

```c
int PyType_CheckExact(PyObject *o)
```
Return true if the object `o` is a type object, but not a subtype of the standard type object. Return false in all other cases.

*New in version 2.2.*

```c
unsigned int PyType_ClearCache()
```
Clear the internal lookup cache. Return the current version tag.

*New in version 2.6.*

```c
void PyType_Modified(PyTypeObject *type)
```
Invalidate the internal lookup cache for the type and all of its subtypes. This function must be called after any manual modification of the attributes or base classes of the type.

*New in version 2.6.*

```c
int PyType_HasFeature(PyObject *o, int feature)
```
Return true if the type object `o` sets the feature `feature`. Type features are denoted by single bit flags.
int PyType_IS_GC(PyObject *o)
    Return true if the type object includes support for the cycle detector; this tests the type flag Py_TPFLAGS_HAVE_GC.

    New in version 2.0.

int PyType_IsSubtype(PyTypeObject *a, PyTypeObject *b)
    Return true if a is a subtype of b.

    New in version 2.2.

This function only checks for actual subtypes, which means that __subclasscheck__() is not called on b. Call PyObject_IsSubclass() to do the same check that issubclass() would do.

PyObject* PyType_GenericAlloc(PyTypeObject *type, Py_ssize_t nitems)

    New in version 2.2.

    Changed in version 2.5: This function used an int type for nitems. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyType_GenericNew(PyTypeObject *type, PyObject *args, PyObject *kwds)

    New in version 2.2.

int PyType_Ready(PyTypeObject *type)
    Finalize a type object. This should be called on all type objects to finish their initialization. This function is responsible for adding inherited slots from a type's base class. Return 0 on success, or return -1 and sets an exception on error.
New in version 2.2.
The None Object

Note that the PyTypeObject for None is not directly exposed in the Python/C API. Since None is a singleton, testing for object identity (using == in C) is sufficient. There is no PyNone_Check() function for the same reason.

PyObject* Py_None
The Python None object, denoting lack of value. This object has no methods. It needs to be treated just like any other object with respect to reference counts.

Py_RETURN_NONE
Properly handle returning Py_None from within a C function.

New in version 2.4.
Plain Integer Objects

**PyIntObject**
This subtype of **PyObject** represents a Python integer object.

**PyTypeObject** **PyInt_Type**
This instance of **PyTypeObject** represents the Python plain integer type. This is the same object as **int** and **types.IntType**.

`int PyInt_Check(PyObject *o)`
Return true if `o` is of type **PyInt_Type** or a subtype of **PyInt_Type**.

*Changed in version 2.2:* Allowed subtypes to be accepted.

`int PyInt_CheckExact(PyObject *o)`
Return true if `o` is of type **PyInt_Type**, but not a subtype of **PyInt_Type**.

*New in version 2.2.*

**PyObject**`* PyInt_FromString(char *str, char **pend, int base)`

*Return value: New reference.*
Return a new **PyIntObject** or **PyLongObject** based on the string value in `str`, which is interpreted according to the radix in `base`. If `pend` is non-`NULL`, `*pend` will point to the first character in `str` which follows the representation of the number. If `base` is 0, the radix will be determined based on the leading characters of `str`: if `str` starts with '0x' or '0X', radix 16 will be used; if `str` starts with '0', radix 8 will be used; otherwise radix 10 will be used. If `base` is not 0, it must be between 2 and 36, inclusive. Leading spaces are ignored. If there are no digits, **ValueError** will be raised. If the string represents a number too large to be contained within the machine’s **long int** type and overflow warnings are being
suppressed, a `PyLongObject` will be returned. If overflow warnings are not being suppressed, `NULL` will be returned in this case.

`PyObject* PyInt_FromLong(long ival)`  
*Return value: New reference.*  
Create a new integer object with a value of `ival`.

The current implementation keeps an array of integer objects for all integers between `-5` and `256`, when you create an int in that range you actually just get back a reference to the existing object. So it should be possible to change the value of `1`. I suspect the behaviour of Python in this case is undefined. :-)

`PyObject* PyInt_FromSsize_t(Py_ssize_t ival)`  
*Return value: New reference.*  
Create a new integer object with a value of `ival`. If the value is larger than `LONG_MAX` or smaller than `LONG_MIN`, a long integer object is returned.  

*New in version 2.5.*

`PyObject* PyInt_FromSize_t(size_t ival)`  
Create a new integer object with a value of `ival`. If the value exceeds `LONG_MAX`, a long integer object is returned.  

*New in version 2.5.*

`long PyInt_AsLong(PyObject *io)`  
Will first attempt to cast the object to a `PyIntObject`, if it is not already one, and then return its value. If there is an error, `-1` is returned, and the caller should check `PyErr_Occurred()` to find out whether there was an error, or whether the value just happened to be `-1`.

`long PyInt_AS_LONG(PyObject *io)`
Return the value of the object \textit{io}. No error checking is performed.

\texttt{unsigned long \textbf{PyInt\_AsUnsignedLongMask}(PyObject *\textit{io})}

Will first attempt to cast the object to a \texttt{PyIntObject} or \texttt{PyLongObject}, if it is not already one, and then return its value as unsigned long. This function does not check for overflow.

\textit{New in version 2.3.}

\texttt{unsigned PY\_LONG\_LONG \textbf{PyInt\_AsUnsignedLongLongMask}(PyObject *\textit{io})}

Will first attempt to cast the object to a \texttt{PyIntObject} or \texttt{PyLongObject}, if it is not already one, and then return its value as unsigned long long, without checking for overflow.

\textit{New in version 2.3.}

\texttt{Py\_ssize\_t \textbf{PyInt\_AsSsize\_t}(PyObject *\textit{io})}

Will first attempt to cast the object to a \texttt{PyIntObject} or \texttt{PyLongObject}, if it is not already one, and then return its value as \texttt{Py\_ssize\_t}.

\textit{New in version 2.5.}

\texttt{long \textbf{PyInt\_GetMax}()}

Return the system's idea of the largest integer it can handle (\texttt{LONG\_MAX}, as defined in the system header files).

\texttt{int \textbf{PyInt\_ClearFreeList}()}

Clear the integer free list. Return the number of items that could not be freed.

\textit{New in version 2.6.}
Boolean Objects

Booleans in Python are implemented as a subclass of integers. There are only two booleans, `Py_False` and `Py_True`. As such, the normal creation and deletion functions don’t apply to booleans. The following macros are available, however.

```c
int PyBool_Check(PyObject *o)
    Return true if o is of type PyBool_Type.
```

*New in version 2.3.*

```c
PyObject* Py_False
    The Python False object. This object has no methods. It needs to be treated just like any other object with respect to reference counts.
```

```c
PyObject* Py_True
    The Python True object. This object has no methods. It needs to be treated just like any other object with respect to reference counts.
```

```c
Py_RETURN_FALSE
    Return Py_False from a function, properly incrementing its reference count.
```

*New in version 2.4.*

```c
Py_RETURN_TRUE
    Return Py_True from a function, properly incrementing its reference count.
```

*New in version 2.4.*

```c
PyObject* PyBool_FromLong(long v)
```
Return a new reference to `Py_True` or `Py_False` depending on the truth value of `v`.

New in version 2.3.
Long Integer Objects

**PyLongObject**
This subtype of `PyObject` represents a Python long integer object.

**PyTypeObject** **PyLong_Type**
This instance of `PyTypeObject` represents the Python long integer type. This is the same object as `long` and `types.LongType`.

```c
int PyLong_Check(PyObject *p)
Return true if its argument is a PyLongObject or a subtype of PyLongObject.
```

*Changed in version 2.2*: Allowed subtypes to be accepted.

```c
int PyLong_CheckExact(PyObject *p)
Return true if its argument is a PyLongObject, but not a subtype of PyLongObject.
```

*New in version 2.2.*

```c
PyObject* PyLong_FromLong(long v)
Return a new PyLongObject object from v, or NULL on failure.
```

```c
PyObject* PyLong_FromUnsignedLong(unsigned long v)
Return a new PyLongObject object from a C unsigned long, or NULL on failure.
```

```c
PyObject* PyLong_FromSsize_t(Py_ssize_t v)
Return a new PyLongObject object from a C Py_ssize_t, or NULL on failure.
```
New in version 2.6.

PyObject* PyLong_FromSize_t(size_t v)
  
  
  Return a new PyLongObject object from a C size_t, or NULL on failure.

New in version 2.6.

PyObject* PyLong_FromLongLong(PY_LONG_LONG v)
  
  
  Return a new PyLongObject object from a C long long, or NULL on failure.

PyObject* PyLong_FromUnsignedLongLong(unsigned PY_LONG_LONG v)
  
  
  Return a new PyLongObject object from a C unsigned long long, or NULL on failure.

PyObject* PyLong_FromDouble(double v)
  
  
  Return a new PyLongObject object from the integer part of v, or NULL on failure.

PyObject* PyLong_FromString(char *str, char **pend, int base)
  
  
  Return a new PyLongObject based on the string value in str, which is interpreted according to the radix in base. If pend is non-NULL, *pend will point to the first character in str which follows the representation of the number. If base is 0, the radix will be determined based on the leading characters of str: if str starts with '0x' or '0X', radix 16 will be used; if str starts with '0', radix 8 will be used; otherwise radix 10 will be used. If base is not 0, it
must be between 2 and 36, inclusive. Leading spaces are ignored. If there are no digits, ValueError will be raised.

PyObject* PyLong_FromUnicode(Py_UNICODE *u, Py_ssize_t length, int base)

    Convert a sequence of Unicode digits to a Python long integer value. The first parameter, u, points to the first character of the Unicode string, length gives the number of characters, and base is the radix for the conversion. The radix must be in the range [2, 36]; if it is out of range, ValueError will be raised.

   New in version 1.6.

   Changed in version 2.5: This function used an int for length. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyLong_FromVoidPtr(void *p)

    Create a Python integer or long integer from the pointer p. The pointer value can be retrieved from the resulting value using PyLong_AsVoidPtr().

   New in version 1.5.2.

   Changed in version 2.5: If the integer is larger than LONG_MAX, a positive long integer is returned.

long PyLong_AsLong(PyObject *pylong)

   Return a C long representation of the contents of pylong. If pylong is greater than LONG_MAX, an OverflowError is raised and -1 will be returned.

long PyLong_AsLongAndOverflow(PyObject *pylong, int *overflow)
Return a C `long` representation of the contents of `pylong`. If `pylong` is greater than `LONG_MAX` or less than `LONG_MIN`, set `*overflow` to 1 or -1, respectively, and return -1; otherwise, set `*overflow` to 0. If any other exception occurs (for example a TypeError or MemoryError), then -1 will be returned and `*overflow` will be 0.

*New in version 2.7.*

**PY_LONG_LONG**

`PyLong_AsLongLongAndOverflow(PyObject *pylong, int *overflow)`

Return a C `long` representation of the contents of `pylong`. If `pylong` is greater than `PY_LLONG_MAX` or less than `PY_LLONG_MIN`, set `*overflow` to 1 or -1, respectively, and return -1; otherwise, set `*overflow` to 0. If any other exception occurs (for example a TypeError or MemoryError), then -1 will be returned and `*overflow` will be 0.

*New in version 2.7.*

`Py_ssize_t PyLong_AsSsize_t(PyObject *pylong)`

Return a C `Py_ssize_t` representation of the contents of `pylong`. If `pylong` is greater than `PY_SSIZE_T_MAX`, an `OverflowError` is raised and -1 will be returned.

*New in version 2.6.*

`unsigned long PyLong_AsUnsignedLong(PyObject *pylong)`

Return a C `unsigned long` representation of the contents of `pylong`. If `pylong` is greater than `ULONG_MAX`, an `OverflowError` is raised.

**PY_LONG_LONG**

`PyLong_AsLongLong(PyObject *pylong)`

Return a C `long long` from a Python long integer. If `pylong`
cannot be represented as a long long, an OverflowError is raised and -1 is returned.

*New in version 2.2.*

unsigned PY_LONG_LONG
PyLong_AsUnsignedLongLong(PyObject *pylong)
Return a C unsigned long long from a Python long integer. If pylong cannot be represented as an unsigned long long, an OverflowError is raised and (unsigned long long)-1 is returned.

*New in version 2.2.*

*Changed in version 2.7:* A negative pylong now raises OverflowError, not TypeError.

unsigned long PyLong_AsUnsignedLongMask(PyObject *io)
Return a C unsigned long from a Python long integer, without checking for overflow.

*New in version 2.3.*

unsigned PY_LONG_LONG
PyLong_AsUnsignedLongLongMask(PyObject *io)
Return a C unsigned long long from a Python long integer, without checking for overflow.

*New in version 2.3.*

double PyLong_AsDouble(PyObject *pylong)
Return a C double representation of the contents of pylong. If pylong cannot be approximately represented as a double, an OverflowError exception is raised and -1.0 will be returned.

void* PyLong_AsVoidPtr(PyObject *pylong)
Convert a Python integer or long integer \textit{pylong} to a C \texttt{void} pointer. If \textit{pylong} cannot be converted, an \texttt{OverflowError} will be raised. This is only assured to produce a usable \texttt{void} pointer for values created with \texttt{PyLong_FromVoidPtr()}.

\textit{New in version 1.5.2.}

\textit{Changed in version 2.5}: For values outside 0..\texttt{LONG_MAX}, both signed and unsigned integers are accepted.
Floating Point Objects

**PyFloatObject**
This subtype of **PyObject** represents a Python floating point object.

**PyTypeObject PyFloat_Type**
This instance of **PyTypeObject** represents the Python floating point type. This is the same object as `float` and `types.FloatType`.

`int PyFloat_Check(PyObject *p)`
Return true if its argument is a `PyFloatObject` or a subtype of `PyFloatObject`.

*Changed in version 2.2:* Allowed subtypes to be accepted.

`int PyFloat_CheckExact(PyObject *p)`
Return true if its argument is a `PyFloatObject`, but not a subtype of `PyFloatObject`.

*New in version 2.2.*

`PyObject* PyFloat_FromString(PyObject *str, char **pend)`
*Return value: New reference.*
Create a `PyFloatObject` object based on the string value in `str`, or `NULL` on failure. The `pend` argument is ignored. It remains only for backward compatibility.

`PyObject* PyFloat_FromDouble(double v)`
*Return value: New reference.*
Create a `PyFloatObject` object from `v`, or `NULL` on failure.

double `PyFloat_AsDouble(PyObject *pyfloat)`
Return a C `double` representation of the contents of `pyfloat`. If `pyfloat` is not a Python floating point object but has a `__float__()` method, this method will first be called to convert `pyfloat` into a float. This method returns `-1.0` upon failure, so one should call `PyErr_Occurred()` to check for errors.

```c
double PyFloat_AS_DOUBLE(PyObject *pyfloat)
```

Return a C `double` representation of the contents of `pyfloat`, but without error checking.

```c
PyObject* PyFloat_GetInfo(void)
```

Return a structseq instance which contains information about the precision, minimum and maximum values of a float. It's a thin wrapper around the header file `float.h`.

`New in version 2.6.`

```c
double PyFloat_GetMax()
```

Return the maximum representable finite float `DBL_MAX` as C `double`.

`New in version 2.6.`

```c
double PyFloat_GetMin()
```

Return the minimum normalized positive float `DBL_MIN` as C `double`.

`New in version 2.6.`

```c
int PyFloat_ClearFreeList()
```

Clear the float free list. Return the number of items that could not be freed.

`New in version 2.6.`
void `PyFloat_AsString`(char *`buf`, `PyFloatObject *v`)

Convert the argument `v` to a string, using the same rules as `str()`. The length of `buf` should be at least 100.

This function is unsafe to call because it writes to a buffer whose length it does not know.

*Deprecated since version 2.7:* Use `PyObject_Str()` or `PyOS_double_to_string()` instead.

void `PyFloat_AsReprString`(char *`buf`, `PyFloatObject *v`)

Same as `PyFloat_AsString`, except uses the same rules as `repr()`. The length of `buf` should be at least 100.

This function is unsafe to call because it writes to a buffer whose length it does not know.

*Deprecated since version 2.7:* Use `PyObject_Repr()` or `PyOS_double_to_string()` instead.
Complex Number Objects

Python's complex number objects are implemented as two distinct types when viewed from the C API: one is the Python object exposed to Python programs, and the other is a C structure which represents the actual complex number value. The API provides functions for working with both.
Complex Numbers as C Structures

Note that the functions which accept these structures as parameters and return them as results do so by value rather than dereferencing them through pointers. This is consistent throughout the API.

**Py_complex**
The C structure which corresponds to the value portion of a Python complex number object. Most of the functions for dealing with complex number objects use structures of this type as input or output values, as appropriate. It is defined as:

```c
typedef struct {
    double real;
    double imag;
} Py_complex;
```

**Py_complex _Py_c_sum(Py_complex left, Py_complex right)**
Return the sum of two complex numbers, using the C `Py_complex` representation.

**Py_complex _Py_c_diff(Py_complex left, Py_complex right)**
Return the difference between two complex numbers, using the C `Py_complex` representation.

**Py_complex _Py_c_neg(Py_complex complex)**
Return the negation of the complex number `complex`, using the C `Py_complex` representation.

**Py_complex _Py_c_prod(Py_complex left, Py_complex right)**
Return the product of two complex numbers, using the C `Py_complex` representation.

**Py_complex _Py_c_quot(Py_complex dividend, Py_complex divisor)**

Return the quotient of two complex numbers, using the C `Py_complex` representation.

If `divisor` is null, this method returns zero and sets `errno` to EDOM.

`Py_complex _Py_c_pow(Py_complex num, Py_complex exp)`

Return the exponentiation of `num` by `exp`, using the C `Py_complex` representation.

If `num` is null and `exp` is not a positive real number, this method returns zero and sets `errno` to EDOM.
Complex Numbers as Python Objects

**PyComplexObject**
This subtype of **PyObject** represents a Python complex number object.

**PyTypeObject PyComplex_Type**
This instance of **PyTypeObject** represents the Python complex number type. It is the same object as complex and types.ComplexType.

```c
int PyComplex_Check(PyObject *p)
```
Return true if its argument is a **PyComplexObject** or a subtype of **PyComplexObject**.

*Changed in version 2.2:* Allowed subtypes to be accepted.

```c
int PyComplex_CheckExact(PyObject *p)
```
Return true if its argument is a **PyComplexObject**, but not a subtype of **PyComplexObject**.

*New in version 2.2.*

**PyObject* PyComplex_FromCComplex(Py_complex v)**
Create a new Python complex number object from a C **Py_complex** value.

**PyObject* PyComplex_FromDoubles(double real, double imag)**
Return a new **PyComplexObject** object from *real* and *imag*.

```c
double PyComplex_RealAsDouble(PyObject *op)
```
Return the real part of \textit{op} as a C \texttt{double}.

\begin{verbatim}
double PyComplex_ImagAsDouble(PyObject \*op)
\end{verbatim}

Return the imaginary part of \textit{op} as a C \texttt{double}.

\begin{verbatim}
Py_complex PyComplex_AsCComplex(PyObject \*op)
\end{verbatim}

Return the \texttt{Py_complex} value of the complex number \textit{op}. Upon failure, this method returns \texttt{-1.0} as a real value.

\textit{Changed in version 2.6}: If \textit{op} is not a Python complex number object but has a \texttt{\_\_complex\_\()} method, this method will first be called to convert \textit{op} to a Python complex number object.
Byte Array Objects

*New in version 2.6.*

**PyByteArrayObject**
This subtype of *PyObject* represents a Python bytearray object.

**PyTypeObject PyByteArray_Type**
This instance of *PyTypeObject* represents the Python bytearray type; it is the same object as *bytearray* in the Python layer.
Type check macros

int PyByteArray_Check(PyObject *o)
    Return true if the object o is a bytearray object or an instance of a subtype of the bytearray type.

int PyByteArray_CheckExact(PyObject *o)
    Return true if the object o is a bytearray object, but not an instance of a subtype of the bytearray type.
### Direct API functions

**PyObject** PyByteArray_FromObject**(PyObject *o)**

Return a new bytearray object from any object, o, that implements the buffer protocol.

**PyObject** PyByteArray_FromStringAndSize**(const char *string, Py_ssize_t len)**

Create a new bytearray object from string and its length, len. On failure, NULL is returned.

**PyObject** PyByteArray_Concat**(PyObject *a, PyObject *b)**

Concat bytearrays a and b and return a new bytearray with the result.

**Py_ssize_t** PyByteArray_Size**(PyObject *bytearray)**

Return the size of bytearray after checking for a NULL pointer.

**char** PyByteArray_AsString**(PyObject *bytearray)**

Return the contents of bytearray as a char array after checking for a NULL pointer.

**int** PyByteArray_Resize**(PyObject *bytearray, Py_ssize_t len)**

Resize the internal buffer of bytearray to len.
Macros

These macros trade safety for speed and they don’t check pointers.

```c
char* PyByteArray_AS_STRING(PyObject *bytearray)
   Macro version of PyByteArray_AsString().
```

```c
Py_ssize_t PyByteArray_GET_SIZE(PyObject *bytearray)
   Macro version of PyByteArray_Size().
```
String/Bytes Objects

These functions raise **TypeError** when expecting a string parameter and are called with a non-string parameter.

**Note:** These functions have been renamed to PyBytes_* in Python 3.x. Unless otherwise noted, the PyBytes functions available in 3.x are aliased to their PyString_* equivalents to help porting.

**PyStringObject**

This subtype of **PyObject** represents a Python string object.

**PyTypeObject PyString_Type**

This instance of **PyTypeObject** represents the Python string type; it is the same object as **str** and **types.StringType** in the Python layer.

```c
int PyString_Check(PyObject *o)
```

Return true if the object o is a string object or an instance of a subtype of the string type.

*Changed in version 2.2:* Allowed subtypes to be accepted.

```c
int PyString_CheckExact(PyObject *o)
```

Return true if the object o is a string object, but not an instance of a subtype of the string type.

*New in version 2.2.*

```c
PyObject* PyString_FromString(const char *v)
```

*Return value: New reference.*

Return a new string object with a copy of the string v as value on success, and **NULL** on failure. The parameter v must not be
NULL; it will not be checked.

PyObject* PyString_FromStringAndSize(const char *v, Py_ssize_t len)

Return a new string object with a copy of the string v as value and length len on success, and NULL on failure. If v is NULL, the contents of the string are uninitialized.

Changed in version 2.5: This function used an int type for len. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyString_FromFormat(const char *format, ...)

Take a C printf()-style format string and a variable number of arguments, calculate the size of the resulting Python string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the format string. The following format characters are allowed:

<table>
<thead>
<tr>
<th>Format Characters</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>%n/a</td>
<td>n/a</td>
<td>The literal % character.</td>
</tr>
<tr>
<td>%c</td>
<td>int</td>
<td>A single character, represented as a C int.</td>
</tr>
<tr>
<td>%d</td>
<td>int</td>
<td>Exactly equivalent to printf(&quot;%d&quot;).</td>
</tr>
<tr>
<td>%u</td>
<td>unsigned int</td>
<td>Exactly equivalent to printf(&quot;%u&quot;).</td>
</tr>
<tr>
<td>%ld</td>
<td>long</td>
<td>Exactly equivalent to printf(&quot;%ld&quot;).</td>
</tr>
<tr>
<td>%lu</td>
<td>unsigned long</td>
<td>Exactly equivalent to printf(&quot;%lu&quot;).</td>
</tr>
<tr>
<td>Format Character</td>
<td>Data Type</td>
<td>Equivalent printf Function</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>%lld</td>
<td>long long</td>
<td>printf(&quot;%lld&quot;)</td>
</tr>
<tr>
<td>%llu</td>
<td>unsigned long long</td>
<td>printf(&quot;%llu&quot;)</td>
</tr>
<tr>
<td>%zd</td>
<td>Py_ssize_t</td>
<td>printf(&quot;%zd&quot;)</td>
</tr>
<tr>
<td>%zu</td>
<td>size_t</td>
<td>printf(&quot;%zu&quot;)</td>
</tr>
<tr>
<td>%i</td>
<td>int</td>
<td>printf(&quot;%i&quot;)</td>
</tr>
<tr>
<td>%x</td>
<td>int</td>
<td>printf(&quot;%x&quot;)</td>
</tr>
<tr>
<td>%s</td>
<td>char*</td>
<td>A null-terminated C character array.</td>
</tr>
<tr>
<td>%p</td>
<td>void*</td>
<td>The hex representation of a C pointer. Mostly equivalent to printf(&quot;%p&quot;) except that it is guaranteed to start with the literal 0x regardless of what the platform’s printf yields.</td>
</tr>
</tbody>
</table>

An unrecognized format character causes all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

**Note:** The “%lld” and “%llu” format specifiers are only available when `HAVE_LONG_LONG` is defined.

*Changed in version 2.7:* Support for “%lld” and “%llu” added.

**PyObject** `PyString_FromFormatV(const char *format, va_list vargs)`

*Return value:* New reference. Identical to `PyString_FromFormat()` except that it takes exactly two arguments.

**Py_ssize_t** `PyString_Size(PyObject *string)`
Return the length of the string in string object *string*.

*Changed in version 2.5:* This function returned an `int` type. This might require changes in your code for properly supporting 64-bit systems.

```c
Py_ssize_t PyString_GET_SIZE(PyObject *string)
```
Macro form of `PyString_Size()` but without error checking.

*Changed in version 2.5:* This macro returned an `int` type. This might require changes in your code for properly supporting 64-bit systems.

```c
char* PyString_AsString(PyObject *string)
```
Return a NUL-terminated representation of the contents of *string*. The pointer refers to the internal buffer of *string*, not a copy. The data must not be modified in any way, unless the string was just created using `PyString_FromStringAndSize(NULL, size)`. It must not be deallocated. If *string* is a Unicode object, this function computes the default encoding of *string* and operates on that. If *string* is not a string object at all, `PyString_AsString()` returns `NULL` and raises `TypeError`.

```c
char* PyString_AS_STRING(PyObject *string)
```
Macro form of `PyString_AsString()` but without error checking. Only string objects are supported; no Unicode objects should be passed.

```c
int PyString_AsStringAndSize(PyObject *obj, char **buffer, Py_ssize_t *length)
```
Return a NUL-terminated representation of the contents of the object *obj* through the output variables `buffer` and `length`.

The function accepts both string and Unicode objects as input. For Unicode objects it returns the default encoded version of the
object. If length is NULL, the resulting buffer may not contain NULL characters; if it does, the function returns -1 and a TypeError is raised.

The buffer refers to an internal string buffer of obj, not a copy. The data must not be modified in any way, unless the string was just created using PyObject_FromStringAndSize(NULL, size). It must not be deallocated. If string is a Unicode object, this function computes the default encoding of string and operates on that. If string is not a string object at all, PyObject_AsStringAndSize() returns -1 and raises TypeError.

Changed in version 2.5: This function used an int * type for length. This might require changes in your code for properly supporting 64-bit systems.

void PyObject_Concat(PyObject **string, PyObject *newpart)
Create a new string object in *string containing the contents of newpart appended to string; the caller will own the new reference. The reference to the old value of string will be stolen. If the new string cannot be created, the old reference to string will still be discarded and the value of *string will be set to NULL; the appropriate exception will be set.

void PyObject_ConcatAndDel(PyObject **string, PyObject *newpart)
Create a new string object in *string containing the contents of newpart appended to string. This version decrements the reference count of newpart.

int _PyObject_Resize(PyObject **string, Py_ssize_t newsize)
A way to resize a string object even though it is “immutable”. Only use this to build up a brand new string object; don’t use this if the string may already be known in other parts of the code. It is an error to call this function if the refcount on the input string object
is not one. Pass the address of an existing string object as an lvalue (it may be written into), and the new size desired. On success, *string holds the resized string object and 0 is returned; the address in *string may differ from its input value. If the reallocation fails, the original string object at *string is deallocated, *string is set to NULL, a memory exception is set, and -1 is returned.

*string* holds the resized string object and 0 is returned; the address in *string may differ from its input value. If the reallocation fails, the original string object at *string is deallocated, *string is set to NULL, a memory exception is set, and -1 is returned.

*string* may differ from its input value. If the reallocation fails, the original string object at *string is deallocated, *string is set to NULL, a memory exception is set, and -1 is returned.

Changed in version 2.5: This function used an int type for newsize. This might require changes in your code for properly supporting 64-bit systems.

**PyObject** * PyString_Format**(PyObject *format, PyObject *args)**

*Return value: New reference.*

Return a new string object from format and args. Analogous to \texttt{format \% args}. The args argument must be a tuple or dict.

**void PyString_InternInPlace**(PyObject **string)**

Intern the argument *string in place. The argument must be the address of a pointer variable pointing to a Python string object. If there is an existing interned string that is the same as *string, it sets *string to it (decrementing the reference count of the old string object and incrementing the reference count of the interned string object), otherwise it leaves *string alone and interns it (incrementing its reference count). (Clarification: even though there is a lot of talk about reference counts, think of this function as reference-count-neutral; you own the object after the call if and only if you owned it before the call.)

**Note:** This function is not available in 3.x and does not have a PyBytes alias.

**PyObject** * PyString_InternFromString**(const char *v)**

*Return value: New reference.*
A combination of `PyString_FromString()` and `PyString_InternInPlace()`, returning either a new string object that has been interned, or a new (“owned”) reference to an earlier interned string object with the same value.

**Note:** This function is not available in 3.x and does not have a PyBytes alias.

```c
PyObject* PyString_Decode(const char *s, Py_ssize_t size, const char *encoding, const char *errors)
```

*Return value: New reference.*

Create an object by decoding size bytes of the encoded buffer `s` using the codec registered for `encoding`. `encoding` and `errors` have the same meaning as the parameters of the same name in the `unicode()` built-in function. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

**Note:** This function is not available in 3.x and does not have a PyBytes alias.

*Changed in version 2.5:* This function used an `int` type for `size`. This might require changes in your code for properly supporting 64-bit systems.

```c
PyObject* PyString_AsDecodedObject(PyObject *str, const char *encoding, const char *errors)
```

*Return value: New reference.*

Decode a string object by passing it to the codec registered for `encoding` and return the result as Python object. `encoding` and `errors` have the same meaning as the parameters of the same name in the string `encode()` method. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.
Note: This function is not available in 3.x and does not have a PyBytes alias.

PyObject* PyString_Encode(const char *s, Py_ssize_t size, const char *encoding, const char *errors)


Encode the char buffer of the given size by passing it to the codec registered for encoding and return a Python object. encoding and errors have the same meaning as the parameters of the same name in the string encode() method. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.

Note: This function is not available in 3.x and does not have a PyBytes alias.

Changed in version 2.5: This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyString_AsEncodedObject(PyObject *str, const char *encoding, const char *errors)


Encode a string object using the codec registered for encoding and return the result as Python object. encoding and errors have the same meaning as the parameters of the same name in the string encode() method. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.

Note: This function is not available in 3.x and does not have a PyBytes alias.
Unicode Objects

Unicode Type

These are the basic Unicode object types used for the Unicode implementation in Python:

**Py_UNICODE**
This type represents the storage type which is used by Python internally as basis for holding Unicode ordinals. Python’s default builds use a 16-bit type for **Py_UNICODE** and store Unicode values internally as UCS2. It is also possible to build a UCS4 version of Python (most recent Linux distributions come with UCS4 builds of Python). These builds then use a 32-bit type for **Py_UNICODE** and store Unicode data internally as UCS4. On platforms where wchar_t is available and compatible with the chosen Python Unicode build variant, **Py_UNICODE** is a typedef alias for wchar_t to enhance native platform compatibility. On all other platforms, **Py_UNICODE** is a typedef alias for either unsigned short (UCS2) or unsigned long (UCS4).

Note that UCS2 and UCS4 Python builds are not binary compatible. Please keep this in mind when writing extensions or interfaces.

**PyUnicodeObject**
This subtype of **PyObject** represents a Python Unicode object.

**PyTypeObject** **PyUnicode_Type**
This instance of **PyTypeObject** represents the Python Unicode type. It is exposed to Python code as **unicode** and **types.UnicodeType**.

The following APIs are really C macros and can be used to do fast
checks and to access internal read-only data of Unicode objects:

```python
int PyUnicode_Check(PyObject *o)
    Return true if the object o is a Unicode object or an instance of a Unicode subtype.

    Changed in version 2.2: Allowed subtypes to be accepted.
```

```python
int PyUnicode_CheckExact(PyObject *o)
    Return true if the object o is a Unicode object, but not an instance of a subtype.

    New in version 2.2.
```

```python
Py_ssize_t PyUnicode_GET_SIZE(PyObject *o)
    Return the size of the object. o has to be a PyUnicodeObject (not checked).

    Changed in version 2.5: This function returned an int type. This might require changes in your code for properly supporting 64-bit systems.
```

```python
Py_ssize_t PyUnicode_GET_DATA_SIZE(PyObject *o)
    Return the size of the object's internal buffer in bytes. o has to be a PyUnicodeObject (not checked).

    Changed in version 2.5: This function returned an int type. This might require changes in your code for properly supporting 64-bit systems.
```

```python
Py_UNICODE* PyUnicode_AS_UNICODE(PyObject *o)
    Return a pointer to the internal Py_UNICODE buffer of the object. o has to be a PyUnicodeObject (not checked).
```

```python
const char* PyUnicode_AS_DATA(PyObject *o)
    Return a pointer to the internal buffer of the object. o has to be a
PyUnicodeObject (not checked).

int PyUnicode_ClearFreeList()
    Clear the free list. Return the total number of freed items.

*New in version 2.6.*

### Unicode Character Properties

Unicode provides many different character properties. The most often needed ones are available through these macros which are mapped to C functions depending on the Python configuration.

int Py_UNICODE_ISSPACE(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a whitespace character.

int Py_UNICODE_ISLOWER(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a lowercase character.

int Py_UNICODE_ISUPPER(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is an uppercase character.

int Py_UNICODE_ISTITLE(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a titlecase character.

int Py_UNICODE_ISLINEBREAK(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a linebreak character.

int Py_UNICODE_ISDECIMAL(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a decimal character.

int Py_UNICODE_ISDIGIT(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a digit character.
int Py_UNICODE_ISNUMERIC(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a numeric character.

int Py_UNICODE_ISALPHA(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is an alphabetic character.

int Py_UNICODE_ISALNUM(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is an alphanumeretic character.

These APIs can be used for fast direct character conversions:

Py_UNICODE Py_UNICODE_TOLOWER(Py_UNICODE ch)
    Return the character ch converted to lower case.

Py_UNICODE Py_UNICODE_TOUPPER(Py_UNICODE ch)
    Return the character ch converted to upper case.

Py_UNICODE Py_UNICODE_TOTITLE(Py_UNICODE ch)
    Return the character ch converted to title case.

int Py_UNICODE_TODECIMAL(Py_UNICODE ch)
    Return the character ch converted to a decimal positive integer. Return -1 if this is not possible. This macro does not raise exceptions.

int Py_UNICODE_TODIGIT(Py_UNICODE ch)
    Return the character ch converted to a single digit integer. Return -1 if this is not possible. This macro does not raise exceptions.

double Py_UNICODE_TONUMERIC(Py_UNICODE ch)
    Return the character ch converted to a double. Return -1.0 if this is not possible. This macro does not raise exceptions.
To create Unicode objects and access their basic sequence properties, use these APIs:

**PyObject** *PyUnicode_FromUnicode*(const Py_UNICODE *u, Py_ssize_t size)

*Return value: New reference.*
Create a Unicode object from the Py_UNICODE buffer `u` of the given size. `u` may be `NULL` which causes the contents to be undefined. It is the user’s responsibility to fill in the needed data. The buffer is copied into the new object. If the buffer is not `NULL`, the return value might be a shared object. Therefore, modification of the resulting Unicode object is only allowed when `u` is `NULL`.

*Changed in version 2.5:* This function used an `int` type for `size`. This might require changes in your code for properly supporting 64-bit systems.

**PyObject** *PyUnicode_FromStringAndSize*(const char *u, Py_ssize_t size)

*Return value: New reference.*
Create a Unicode object from the char buffer `u`. The bytes will be interpreted as being UTF-8 encoded. `u` may also be `NULL` which causes the contents to be undefined. It is the user’s responsibility to fill in the needed data. The buffer is copied into the new object. If the buffer is not `NULL`, the return value might be a shared object. Therefore, modification of the resulting Unicode object is only allowed when `u` is `NULL`.

*New in version 2.6.*

**PyObject** *PyUnicode_FromString*(const char *u)

*Return value: New reference.*
Create a Unicode object from a UTF-8 encoded null-terminated
char buffer \textit{u}.

\textit{New in version 2.6.}

\textbf{PyObject* \texttt{PyUnicode_FromFormat}}(const char *\textit{format}, ...)

\textit{Return value: New reference.}

Take a C \texttt{printf()}-style \textit{format} string and a variable number of arguments, calculate the size of the resulting Python unicode string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the \textit{format} string. The following format characters are allowed:

<table>
<thead>
<tr>
<th>Format Characters</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>%%</td>
<td>\textit{n/a}</td>
<td>The literal % character.</td>
</tr>
<tr>
<td>%c</td>
<td>\textit{int}</td>
<td>A single character, represented as a C int.</td>
</tr>
<tr>
<td>%d</td>
<td>\textit{int}</td>
<td>Exactly equivalent to printf(&quot;%d&quot;).</td>
</tr>
<tr>
<td>%u</td>
<td>unsigned \textit{int}</td>
<td>Exactly equivalent to printf(&quot;%u&quot;).</td>
</tr>
<tr>
<td>%ld</td>
<td>\textit{long}</td>
<td>Exactly equivalent to printf(&quot;%ld&quot;).</td>
</tr>
<tr>
<td>%lu</td>
<td>unsigned \textit{long}</td>
<td>Exactly equivalent to printf(&quot;%lu&quot;).</td>
</tr>
<tr>
<td>%zd</td>
<td>\textit{Py_ssize_t}</td>
<td>Exactly equivalent to printf(&quot;%zd&quot;).</td>
</tr>
<tr>
<td>%zu</td>
<td>\textit{size_t}</td>
<td>Exactly equivalent to printf(&quot;%zu&quot;).</td>
</tr>
<tr>
<td>%i</td>
<td>\textit{int}</td>
<td>Exactly equivalent to printf(&quot;%i&quot;).</td>
</tr>
<tr>
<td>%x</td>
<td>\textit{int}</td>
<td>Exactly equivalent to printf(&quot;%x&quot;).</td>
</tr>
</tbody>
</table>
| \%s                | char*          | A null-terminated C
<table>
<thead>
<tr>
<th>Character</th>
<th>Argument Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%p</code></td>
<td>void*</td>
<td>void*</td>
</tr>
<tr>
<td><code>%u</code></td>
<td>PyObject*</td>
<td>A unicode object.</td>
</tr>
<tr>
<td><code>%v</code></td>
<td>PyObject*, char *</td>
<td>A unicode object (which may be <code>NULL</code>) and a null-terminated C character array as a second parameter (which will be used, if the first parameter is <code>NULL</code>).</td>
</tr>
<tr>
<td><code>%S</code></td>
<td>PyObject*</td>
<td>The result of calling <code>PyObject_Unicode()</code>.</td>
</tr>
<tr>
<td><code>%R</code></td>
<td>PyObject*</td>
<td>The result of calling <code>PyObject_Repr()</code>.</td>
</tr>
</tbody>
</table>

An unrecognized format character causes all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

*New in version 2.6.*

```c
PyObject* PyUnicode_FromFormatV(const char *format, va_list vars)
```

*Return value: New reference.*

Identical to `PyUnicode_FromFormat()` except that it takes exactly two arguments.

*New in version 2.6.*

```c
Py_UNICODE* PyUnicode_AsUnicode(PyObject *unicode)
```

Return a read-only pointer to the Unicode object's internal
**Py_UNICODE** buffer, *NULL* if *unicode* is not a Unicode object. Note that the resulting **Py_UNICODE** string may contain embedded null characters, which would cause the string to be truncated when used in most C functions.

**Py_ssize_t** **PyUnicode_GetSize**(PyObject **unicode*)
Return the length of the Unicode object.

*Changed in version 2.5:* This function returned an int type. This might require changes in your code for properly supporting 64-bit systems.

**PyObject** *PyUnicode_FromEncodedObject**(PyObject *obj, const char *encoding, const char *errors)
*Return value: New reference.*
Coerce an encoded object *obj* to a Unicode object and return a reference with incremented refcount.

String and other char buffer compatible objects are decoded according to the given encoding and using the error handling defined by errors. Both can be *NULL* to have the interface use the default values (see the next section for details).

All other objects, including Unicode objects, cause a TypeError to be set.

The API returns *NULL* if there was an error. The caller is responsible for deallocating the returned objects.

**PyObject** *PyUnicode_FromObject**(PyObject *obj)
*Return value: New reference.*
Shortcut for PyUnicode_FromEncodedObject(*obj, NULL, "strict") which is used throughout the interpreter whenever coercion to Unicode is needed.
If the platform supports wchar_t and provides a header file wchar.h, Python can interface directly to this type using the following functions. Support is optimized if Python’s own Py_UNICODE type is identical to the system’s wchar_t.

## wchar_t Support

wchar_t support for platforms which support it:

**PyObject**\* PyUnicode_FromWideChar(const wchar_t *w, Py_ssize_t size)

*Return value: New reference.*
Create a Unicode object from the wchar_t buffer w of the given size. Return NULL on failure.

*Changed in version 2.5:* This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.

Py_ssize_t PyUnicode_AsWideChar(PyUnicodeObject *unicode, wchar_t *w, Py_ssize_t size)

Copy the Unicode object contents into the wchar_t buffer w. At most size wchar_t characters are copied (excluding a possibly trailing 0-termination character). Return the number of wchar_t characters copied or -1 in case of an error. Note that the resulting wchar_t string may or may not be 0-terminated. It is the responsibility of the caller to make sure that the wchar_t string is 0-terminated in case this is required by the application. Also, note that the wchar_t* string might contain null characters, which would cause the string to be truncated when used with most C functions.

*Changed in version 2.5:* This function returned an int type and
used an `int` type for `size`. This might require changes in your code for properly supporting 64-bit systems.
Built-in Codecs

Python provides a set of built-in codecs which are written in C for speed. All of these codecs are directly usable via the following functions.

Many of the following APIs take two arguments encoding and errors, and they have the same semantics as the ones of the built-in `unicode()` Unicode object constructor.

Setting encoding to `NULL` causes the default encoding to be used which is ASCII. The file system calls should use `Py_FileSystemDefaultEncoding` as the encoding for file names. This variable should be treated as read-only: on some systems, it will be a pointer to a static string, on others, it will change at run-time (such as when the application invokes `setlocale()`).

Error handling is set by errors which may also be set to `NULL` meaning to use the default handling defined for the codec. Default error handling for all built-in codecs is “strict” (`ValueError` is raised).

The codecs all use a similar interface. Only deviation from the following generic ones are documented for simplicity.

Generic Codecs

These are the generic codec APIs:

```c
PyObject* PyUnicode_Decode(const char *s, Py_ssize_t size, const char *encoding, const char *errors)
```

*Return value: New reference.*

Create a Unicode object by decoding `size` bytes of the encoded string `s`. `encoding` and `errors` have the same meaning as the parameters of the same name in the `unicode()` built-in function.
The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.

*Changed in version 2.5:* This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.

`PyObject* PyUnicode_Encode(const Py_UNICODE *s, Py_ssize_t size, const char *encoding, const char *errors)`

*Return value: New reference.*

Encode the Py_UNICODE buffer s of the given size and return a Python string object. encoding and errors have the same meaning as the parameters of the same name in the Unicode encode() method. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.

*Changed in version 2.5:* This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.

`PyObject* PyUnicode_AsEncodedString(PyObject *unicode, const char *encoding, const char *errors)`

*Return value: New reference.*

Encode a Unicode object and return the result as Python string object. encoding and errors have the same meaning as the parameters of the same name in the Unicode encode() method. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.

### UTF-8 Codecs

These are the UTF-8 codec APIs:

`PyObject* PyUnicode_DecodeUTF8(const char *s, Py_ssize_t size,`
const char *errors)

Create a Unicode object by decoding size bytes of the UTF-8 encoded string s. Return NULL if an exception was raised by the codec.

Changed in version 2.5: This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyUnicode_DecodeUTF8Stateful(const char *s, Py_ssize_t size, const char *errors, Pysize_t *consumed)

If consumed is NULL, behave like PyUnicode_DecodeUTF8(). If consumed is not NULL, trailing incomplete UTF-8 byte sequences will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in consumed.

New in version 2.4.

Changed in version 2.5: This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyUnicode_EncodeUTF8(const Py_UNICODE *s, Py_ssize_t size, const char *errors)

Encode the Py_UNICODE buffer s of the given size using UTF-8 and return a Python string object. Return NULL if an exception was raised by the codec.

Changed in version 2.5: This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.
PyObject* PyUnicode_AsUTF8String(PyObject *unicode)


Encode a Unicode object using UTF-8 and return the result as Python string object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

UTF-32 Codecs

These are the UTF-32 codec APIs:

PyObject* PyUnicode_DecodeUTF32(const char *s, Py_ssize_t size, const char *errors, int *byteorder)

Decide size bytes from a UTF-32 encoded buffer string and return the corresponding Unicode object. errors (if non-NULL) defines the error handling. It defaults to “strict”.

If byteorder is non-NULL, the decoder starts decoding using the given byte order:

*byteorder == -1: little endian
*byteorder == 0: native order
*byteorder == 1: big endian

If *byteorder is zero, and the first four bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If *byteorder is -1 or 1, any byte order mark is copied to the output.

After completion, *byteorder is set to the current byte order at the end of input data.

In a narrow build code points outside the BMP will be decoded as surrogate pairs.

If byteorder is NULL, the codec starts in native order mode.
Return *NULL* if an exception was raised by the codec.

*New in version 2.6.*

```c
PyObject* PyUnicode_DecodeUTF32Stateful(const char *s,
Py_ssize_t size, const char *errors, int *byteorder,
Py_ssize_t *consumed)
```

If `consumed` is *NULL*, behave like `PyUnicode_DecodeUTF32()`. If `consumed` is not *NULL*, `PyUnicode_DecodeUTF32Stateful()` will not treat trailing incomplete UTF-32 byte sequences (such as a number of bytes not divisible by four) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in `consumed`.

*New in version 2.6.*

```c
PyObject* PyUnicode_EncodeUTF32(const Py_UNICODE *s,
Py_ssize_t size, const char *errors, int byteorder)
```

Return a Python bytes object holding the UTF-32 encoded value of the Unicode data in `s`. Output is written according to the following byte order:

<table>
<thead>
<tr>
<th>byteorder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>little endian</td>
</tr>
<tr>
<td>0</td>
<td>native byte order (writes a BOM mark)</td>
</tr>
<tr>
<td>1</td>
<td>big endian</td>
</tr>
</tbody>
</table>

If `byteorder` is 0, the output string will always start with the Unicode BOM mark (U+FEFF). In the other two modes, no BOM mark is prepended.

If `Py_UNICODE_WIDE` is not defined, surrogate pairs will be output as a single code point.

Return *NULL* if an exception was raised by the codec.

*New in version 2.6.*
PyObject* PyUnicode_AsUTF32String(PyObject *unicode)
Return a Python string using the UTF-32 encoding in native byte order. The string always starts with a BOM mark. Error handling is “strict”. Return NULL if an exception was raised by the codec.

New in version 2.6.

UTF-16 Codecs

These are the UTF-16 codec APIs:

PyObject* PyUnicode_DecodeUTF16(const char *s, Py_ssize_t size, const char *errors, int *byteorder)
Decode size bytes from a UTF-16 encoded buffer string and return the corresponding Unicode object. errors (if non-NULL) defines the error handling. It defaults to “strict”.

If byteorder is non-NULL, the decoder starts decoding using the given byte order:

*byteorder == -1: little endian
*byteorder == 0: native order
*byteorder == 1: big endian

If *byteorder is zero, and the first two bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If *byteorder is -1 or 1, any byte order mark is copied to the output (where it will result in either a \u00ef or a \ufffe character).

After completion, *byteorder is set to the current byte order at the end of input data.

If byteorder is NULL, the codec starts in native order mode.
Return NULL if an exception was raised by the codec.

*Changed in version 2.5:* This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.

```
PyObject* PyUnicode_DecodeUTF16Stateful(const char *s,
Py_ssize_t size, const char *errors, int *byteorder,
Py_ssize_t *consumed)

If consumed is NULL, behave like PyUnicode_DecodeUTF16(). If consumed is not NULL, PyUnicode_DecodeUTF16Stateful() will not treat trailing incomplete UTF-16 byte sequences (such as an odd number of bytes or a split surrogate pair) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in consumed.

New in version 2.4.

*Changed in version 2.5:* This function used an int type for size and an int * type for consumed. This might require changes in your code for properly supporting 64-bit systems.

```

```
PyObject* PyUnicode_EncodeUTF16(const Py_UNICODE *s,
Py_ssize_t size, const char *errors, int byteorder)

Return a Python string object holding the UTF-16 encoded value of the Unicode data in s. Output is written according to the following byte order:

<table>
<thead>
<tr>
<th>byteorder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>little endian</td>
</tr>
<tr>
<td>0</td>
<td>native byte order (writes a BOM mark)</td>
</tr>
<tr>
<td>1</td>
<td>big endian</td>
</tr>
</tbody>
</table>

If byteorder is 0, the output string will always start with the Unicode BOM mark (U+FEFF). In the other two modes, no BOM
mark is prepended.

If `Py_UNICODE_WIDE` is defined, a single `Py_UNICODE` value may get represented as a surrogate pair. If it is not defined, each `Py_UNICODE` values is interpreted as a UCS-2 character.

Return `NULL` if an exception was raised by the codec.

*Changed in version 2.5:* This function used an `int` type for `size`. This might require changes in your code for properly supporting 64-bit systems.

```c
PyObject* PyUnicode_AsUTF16String(PyObject *unicode)

Return a Python string using the UTF-16 encoding in native byte order. The string always starts with a BOM mark. Error handling is “strict”. Return `NULL` if an exception was raised by the codec.
```

### UTF-7 Codecs

These are the UTF-7 codec APIs:

```c
PyObject* PyUnicode_DecodeUTF7(const char *s, Py_ssize_t size, const char *errors)

Create a Unicode object by decoding `size` bytes of the UTF-7 encoded string `s`. Return `NULL` if an exception was raised by the codec.
```

```c
PyObject* PyUnicode_DecodeUTF7Stateful(const char *s, Py_ssize_t size, const char *errors, Py_ssize_t *consumed)

If `consumed` is `NULL`, behave like `PyUnicode_DecodeUTF7()`. If `consumed` is not `NULL`, trailing incomplete UTF-7 base-64 sections will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in `consumed`.
```
PyObject* PyUnicode_EncodeUTF7(const Py_UNICODE *s,
Py_ssize_t size, int base64SetO, int base64WhiteSpace, const char *errors)

Encode the Py_UNICODE buffer of the given size using UTF-7 and return a Python bytes object. Return NULL if an exception was raised by the codec.

If base64SetO is nonzero, “Set O” (punctuation that has no otherwise special meaning) will be encoded in base-64. If base64WhiteSpace is nonzero, whitespace will be encoded in base-64. Both are set to zero for the Python “utf-7” codec.

Unicode-Escape Codecs

These are the “Unicode Escape” codec APIs:

PyObject* PyUnicode_DecodeUnicodeEscape(const char *s,
Py_ssize_t size, const char *errors)

Create a Unicode object by decoding size bytes of the Unicode-Escape encoded string s. Return NULL if an exception was raised by the codec.

Changed in version 2.5: This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyUnicode_EncodeUnicodeEscape(const
Py_UNICODE *s, Py_ssize_t size)

Encode the Py_UNICODE buffer of the given size using Unicode-Escape and return a Python string object. Return NULL if an exception was raised by the codec.

Changed in version 2.5: This function used an int type for size.
This might require changes in your code for properly supporting 64-bit systems.

**PyObject* `PyUnicode_AsUnicodeEscapeString(PyObject *unicode)`**

*Return value: New reference.*

Encode a Unicode object using Unicode-Escape and return the result as Python string object. Error handling is “strict”. Return `NULL` if an exception was raised by the codec.

---

**Raw-Unicode-Escape Codecs**

These are the “Raw Unicode Escape” codec APIs:

**PyObject* `PyUnicode_DecodeRawUnicodeEscape(const char *s, Py_ssize_t size, const char *errors)`**

*Return value: New reference.*

Create a Unicode object by decoding size bytes of the Raw-Unicode-Escape encoded string s. Return `NULL` if an exception was raised by the codec.

*Changed in version 2.5:* This function used an `int` type for `size`. This might require changes in your code for properly supporting 64-bit systems.

**PyObject* `PyUnicode_EncodeRawUnicodeEscape(const Py_UNICODE *s, Py_ssize_t size, const char *errors)`**

*Return value: New reference.*

Encode the `Py_UNICODE` buffer of the given `size` using Raw-Unicode-Escape and return a Python string object. Return `NULL` if an exception was raised by the codec.

*Changed in version 2.5:* This function used an `int` type for `size`. This might require changes in your code for properly supporting 64-bit systems.
PyObject*
PyUnicode_AsRawUnicodeEscapeString(PyObject *unicode)

*Return value: New reference.*

Encode a Unicode object using Raw-Unicode-Escape and return the result as Python string object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

## Latin-1 Codecs

These are the Latin-1 codec APIs: Latin-1 corresponds to the first 256 Unicode ordinals and only these are accepted by the codecs during encoding.

PyObject* PyUnicode_DecodeLatin1(const char *s, Py_ssize_t size, const char *errors)

*Return value: New reference.*

Create a Unicode object by decoding size bytes of the Latin-1 encoded string s. Return NULL if an exception was raised by the codec.

*Changed in version 2.5:* This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyUnicode_EncodeLatin1(const Py_UNICODE *s, Py_ssize_t size, const char *errors)

*Return value: New reference.*

Encode the Py_UNICODE buffer of the given size using Latin-1 and return a Python string object. Return NULL if an exception was raised by the codec.

*Changed in version 2.5:* This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.
PyObject* **PyUnicode_AsLatin1String**(PyObject *unicode)

*Return value: New reference.*

Encode a Unicode object using Latin-1 and return the result as Python string object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

### ASCII Codecs

These are the ASCII codec APIs. Only 7-bit ASCII data is accepted. All other codes generate errors.

PyObject* **PyUnicode_DecodeASCII**(const char *s, Py_ssize_t size, const char *errors)

*Return value: New reference.*

Create a Unicode object by decoding size bytes of the ASCII encoded string s. Return NULL if an exception was raised by the codec.

*Changed in version 2.5:* This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.

PyObject* **PyUnicode_EncodeASCII**(const Py_UNICODE *s, Py_ssize_t size, const char *errors)

*Return value: New reference.*

Encode the Py_UNICODE buffer of the given size using ASCII and return a Python string object. Return NULL if an exception was raised by the codec.

*Changed in version 2.5:* This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.

PyObject* **PyUnicode_AsASCIIString**(PyObject *unicode)

*Return value: New reference.*
Encode a Unicode object using ASCII and return the result as Python string object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

**Character Map Codecs**

This codec is special in that it can be used to implement many different codecs (and this is in fact what was done to obtain most of the standard codecs included in the `encodings` package). The codec uses mapping to encode and decode characters.

Decoding mappings must map single string characters to single Unicode characters, integers (which are then interpreted as Unicode ordinals) or None (meaning “undefined mapping” and causing an error).

Encoding mappings must map single Unicode characters to single string characters, integers (which are then interpreted as Latin-1 ordinals) or None (meaning “undefined mapping” and causing an error).

The mapping objects provided must only support the `__getitem__` mapping interface.

If a character lookup fails with a LookupError, the character is copied as-is meaning that its ordinal value will be interpreted as Unicode or Latin-1 ordinal resp. Because of this, mappings only need to contain those mappings which map characters to different code points.

These are the mapping codec APIs:

```c
PyObject* PyUnicode_DecodeCharmap(const char *s, Py_ssize_t size, PyObject *mapping, const char *errors)
```

*Return value: New reference.*

Create a Unicode object by decoding `size` bytes of the encoded
string s using the given \textit{mapping} object. Return \textit{NULL} if an exception was raised by the codec. If \textit{mapping} is \textit{NULL} latin-1 decoding will be done. Else it can be a dictionary mapping byte or a unicode string, which is treated as a lookup table. Byte values greater that the length of the string and U+FFFE “characters” are treated as “undefined mapping”.

\textit{Changed in version 2.4:} Allowed unicode string as mapping argument.

\textit{Changed in version 2.5:} This function used an \texttt{int} type for \texttt{size}. This might require changes in your code for properly supporting 64-bit systems.

\texttt{PyObject* PyUnicode\_EncodeCharmap(const Py\_UNICODE *s, Py\_ssize\_t size, PyObject *mapping, const char *errors)}

\textit{Return value: New reference.}

Encode the \texttt{Py\_UNICODE} buffer of the given \texttt{size} using the given \textit{mapping} object and return a Python string object. Return \textit{NULL} if an exception was raised by the codec.

\textit{Changed in version 2.5:} This function used an \texttt{int} type for \texttt{size}. This might require changes in your code for properly supporting 64-bit systems.

\texttt{PyObject* PyUnicode\_AsCharmapString(PyObject *unicode, PyObject *mapping)}

\textit{Return value: New reference.}

Encode a Unicode object using the given \textit{mapping} object and return the result as Python string object. Error handling is “strict”. Return \textit{NULL} if an exception was raised by the codec.

The following codec API is special in that maps Unicode to Unicode.

\texttt{PyObject* PyUnicode\_TranslateCharmap(const Py\_UNICODE *s, Py\_ssize\_t size, PyObject *table, const char *errors)}
**Return value: New reference.**
Translate a `Py_UNICODE` buffer of the given `size` by applying a character mapping `table` to it and return the resulting Unicode object. Return `NULL` when an exception was raised by the codec.

The `mapping` table must map Unicode ordinal integers to Unicode ordinal integers or `None` (causing deletion of the character).

Mapping tables need only provide the `__getitem__()` interface; dictionaries and sequences work well. Unmapped character ordinals (ones which cause a `LookupError`) are left untouched and are copied as-is.

**Changed in version 2.5:** This function used an `int` type for `size`. This might require changes in your code for properly supporting 64-bit systems.

---

**MBCS codecs for Windows**

These are the MBCS codec APIs. They are currently only available on Windows and use the Win32 MBCS converters to implement the conversions. Note that MBCS (or DBCS) is a class of encodings, not just one. The target encoding is defined by the user settings on the machine running the codec.

```c
PyObject* PyUnicode_DecodeMBCS(const char *s, Py_ssize_t size,
const char *errors)
```

**Return value: New reference.**
Create a Unicode object by decoding `size` bytes of the MBCS encoded string `s`. Return `NULL` if an exception was raised by the codec.

**Changed in version 2.5:** This function used an `int` type for `size`. This might require changes in your code for properly supporting
64-bit systems.

**PyObject** `PyUnicode_DecodeMBCSStateful` (const char *s, int size, const char *errors, int *consumed)

If consumed is NULL, behave like `PyUnicode_DecodeMBCS()`. If consumed is not NULL, `PyUnicode_DecodeMBCSStateful()` will not decode trailing lead byte and the number of bytes that have been decoded will be stored in consumed.

*New in version 2.5.*

**PyObject** `PyUnicode_EncodeMBCS` (const Py_UNICODE *s, Py_ssize_t size, const char *errors)

*Return value: New reference.*

Encode the Py_UNICODE buffer of the given size using MBCS and return a Python string object. Return NULL if an exception was raised by the codec.

*Changed in version 2.5:* This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.

**PyObject** `PyUnicode_AsMBCSString` (PyObject *unicode)

*Return value: New reference.*

Encode a Unicode object using MBCS and return the result as Python string object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

---

**Methods & Slots**
Methods and Slot Functions

The following APIs are capable of handling Unicode objects and strings on input (we refer to them as strings in the descriptions) and return Unicode objects or integers as appropriate.

They all return *NULL* or -1 if an exception occurs.

**PyObject**\* **PyUnicode_Concat**\(\text{PyObject} *\text{left}, \text{PyObject} *\text{right}\)

*Return value: New reference.*

Concat two strings giving a new Unicode string.

**PyObject**\* **PyUnicode_Split**\(\text{PyObject} *\text{s}, \text{PyObject} *\text{sep},\)

\(\text{Py_ssize_t} \text{maxsplit}\)

*Return value: New reference.*

Split a string giving a list of Unicode strings. If \text{sep} is *NULL*, splitting will be done at all whitespace substrings. Otherwise, splits occur at the given separator. At most \text{maxsplit} splits will be done. If negative, no limit is set. Separators are not included in the resulting list.

*Changed in version 2.5:* This function used an int type for \text{maxsplit}. This might require changes in your code for properly supporting 64-bit systems.

**PyObject**\* **PyUnicode_Splitlines**\(\text{PyObject} *\text{s}, \text{int} \text{keepend}\)

*Return value: New reference.*

Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. If \text{keepend} is 0, the Line break characters are not included in the resulting strings.

**PyObject**\* **PyUnicode_Translate**\(\text{PyObject} *\text{str}, \text{PyObject} *\text{table},\)

\(\text{const} \text{char} *\text{errors}\)

*Return value: New reference.*
Translate a string by applying a character mapping table to it and return the resulting Unicode object.

The mapping table must map Unicode ordinal integers to Unicode ordinal integers or None (causing deletion of the character).

Mapping tables need only provide the __getitem__() interface; dictionaries and sequences work well. Unmapped character ordinals (ones which cause a LookupError) are left untouched and are copied as-is.

errors has the usual meaning for codecs. It may be NULL which indicates to use the default error handling.

PyObject* PyUnicode_Join(PyObject *separator, PyObject *seq)

Join a sequence of strings using the given separator and return the resulting Unicode string.

Py_ssize_t PyUnicode_Tailmatch(PyObject *str, PyObject *substr, Py_ssize_t start, Py_ssize_t end, int direction)

Return 1 if substr matches str[start:end] at the given tail end (direction == -1 means to do a prefix match, direction == 1 a suffix match), 0 otherwise. Return -1 if an error occurred.

Changed in version 2.5: This function used an int type for start and end. This might require changes in your code for properly supporting 64-bit systems.

Py_ssize_t PyUnicode_Find(PyObject *str, PyObject *substr, Py_ssize_t start, Py_ssize_t end, int direction)

Return the first position of substr in str[start:end] using the given direction (direction == 1 means to do a forward search, direction == -1 a backward search). The return value is the index
of the first match; a value of \(-1\) indicates that no match was found, and \(-2\) indicates that an error occurred and an exception has been set.

*Changed in version 2.5:* This function used an int type for \(\text{start}\) and \(\text{end}\). This might require changes in your code for properly supporting 64-bit systems.

```c
Py_ssize_t PyUnicode_Count(PyObject *str, PyObject *substr, Py_ssize_t start, Py_ssize_t end)
```

Return the number of non-overlapping occurrences of \(\text{substr}\) in \(\text{str}[\text{start}:\text{end}]\). Return \(-1\) if an error occurred.

*Changed in version 2.5:* This function returned an int type and used an int type for \(\text{start}\) and \(\text{end}\). This might require changes in your code for properly supporting 64-bit systems.

```c
PyObject* PyUnicode_REPLACE(PyObject *str, PyObject *substr, PyObject *replstr, Py_ssize_t maxcount)
```

*Return value: New reference.*

Replace at most \(\text{maxcount}\) occurrences of \(\text{substr}\) in \(\text{str}\) with \(\text{replstr}\) and return the resulting Unicode object. \(\text{maxcount} == -1\) means replace all occurrences.

*Changed in version 2.5:* This function used an int type for \(\text{maxcount}\). This might require changes in your code for properly supporting 64-bit systems.

```c
int PyUnicode_Compare(PyObject *left, PyObject *right)
```

Compare two strings and return \(-1\), \(0\), \(1\) for less than, equal, and greater than, respectively.

```c
int PyUnicode_RichCompare(PyObject *left, PyObject *right, int op)
```

Rich compare two unicode strings and return one of the following:
- **NULL** in case an exception was raised
- **Py_True** or **Py_False** for successful comparisons
- **Py_NotImplemented** in case the type combination is unknown

Note that **Py_EQ** and **Py_NE** comparisons can cause a `UnicodeWarning` in case the conversion of the arguments to Unicode fails with a `UnicodeDecodeError`.

Possible values for `op` are **Py_GT**, **Py_GE**, **Py_EQ**, **Py_NE**, **Py_LT**, and **Py_LE**.

**PyObject** * PyUnicode_Format (PyObject *format, PyObject *args) *

*Return value: New reference.*

Return a new string object from `format` and `args`; this is analogous to `format % args`.

**int** PyUnicode_Contains (PyObject *container, PyObject *element) *

Check whether `element` is contained in `container` and return true or false accordingly.

`element` has to coerce to a one element Unicode string. -1 is returned if there was an error.
Buffers and Memoryview Objects

Python objects implemented in C can export a group of functions called the “buffer interface.” These functions can be used by an object to expose its data in a raw, byte-oriented format. Clients of the object can use the buffer interface to access the object data directly, without needing to copy it first.

Two examples of objects that support the buffer interface are strings and arrays. The string object exposes the character contents in the buffer interface’s byte-oriented form. An array can only expose its contents via the old-style buffer interface. This limitation does not apply to Python 3, where memoryview objects can be constructed from arrays, too. Array elements may be multi-byte values.

An example user of the buffer interface is the file object's write() method. Any object that can export a series of bytes through the buffer interface can be written to a file. There are a number of format codes to PyArg_ParseTuple() that operate against an object's buffer interface, returning data from the target object.

Starting from version 1.6, Python has been providing Python-level buffer objects and a C-level buffer API so that any built-in or used-defined type can expose its characteristics. Both, however, have been deprecated because of various shortcomings, and have been officially removed in Python 3 in favour of a new C-level buffer API and a new Python-level object named memoryview.

The new buffer API has been backported to Python 2.6, and the memoryview object has been backported to Python 2.7. It is strongly advised to use them rather than the old APIs, unless you are blocked from doing so for compatibility reasons.
The new-style Py_buffer struct

Py_buffer
    void *buf
    A pointer to the start of the memory for the object.

Py_ssize_t len
    The total length of the memory in bytes.

int readonly
    An indicator of whether the buffer is read only.

const char *format
    A NULL terminated string in struct module style syntax giving the contents of the elements available through the buffer. If this is NULL, "B" (unsigned bytes) is assumed.

int ndim
    The number of dimensions the memory represents as a multi-dimensional array. If it is 0, strides and suboffsets must be NULL.

Py_ssize_t *shape
    An array of Py_ssize_t the length of ndim giving the shape of the memory as a multi-dimensional array. Note that ((*shape)[0] * ... * (*shape)[ndims-1])*itemsize should be equal to len.

Py_ssize_t *strides
    An array of Py_ssize_t the length of ndim giving the number of bytes to skip to get to a new element in each dimension.

Py_ssize_t *suboffsets
An array of `Py_ssize_t` the length of `ndim`. If these suboffset numbers are greater than or equal to 0, then the value stored along the indicated dimension is a pointer and the suboffset value dictates how many bytes to add to the pointer after de-referencing. A suboffset value that is negative indicates that no de-referencing should occur (striding in a contiguous memory block).

If all suboffsets are negative (i.e. no de-referencing is needed, then this field must be NULL (the default value).

Here is a function that returns a pointer to the element in an N-D array pointed to by an N-dimensional index when there are both non-NULL strides and suboffsets:

```c
void *get_item_pointer(int ndim, void *buf, Py_ssize_t *strides, Py_ssize_t *suboffsets, Py_ssize_t *indices) {
    char *pointer = (char*)buf;
    int i;
    for (i = 0; i < ndim; i++) {
        pointer += strides[i] * indices[i];
        if (suboffsets[i] >= 0) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}
```

**Py_ssize_t itemsize**

This is a storage for the itemsize (in bytes) of each element of the shared memory. It is technically un-necessary as it can be obtained using `PyBuffer_SizeFromFormat()`, however an exporter may know this information without parsing the format string and it is necessary to know the itemsize for proper interpretation of striding. Therefore, storing it is more convenient and faster.
void *internal

This is for use internally by the exporting object. For example, this might be re-cast as an integer by the exporter and used to store flags about whether or not the shape, strides, and suboffsets arrays must be freed when the buffer is released. The consumer should never alter this value.
Buffer related functions

int **buffer_check(PyObject *obj)**
Return 1 if *obj* supports the buffer interface otherwise 0.

int **buffer_get(PyObject *obj, Py_buffer *view, int flags)**
Export *obj* into a **buffer**, *view*. These arguments must never be NULL. The *flags* argument is a bit field indicating what kind of buffer the caller is prepared to deal with and therefore what kind of buffer the exporter is allowed to return. The buffer interface allows for complicated memory sharing possibilities, but some caller may not be able to handle all the complexity but may want to see if the exporter will let them take a simpler view to its memory.

Some exporters may not be able to share memory in every possible way and may need to raise errors to signal to some consumers that something is just not possible. These errors should be a **buffer_error** unless there is another error that is actually causing the problem. The exporter can use flags information to simplify how much of the **buffer** structure is filled in with non-default values and/or raise an error if the object can’t support a simpler view of its memory.

0 is returned on success and -1 on error.

The following table gives possible values to the *flags* arguments.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>buffer_simply</strong></td>
<td>This is the default flag state. The returned buffer may or may not have writable memory. The format of the data will be assumed to be unsigned bytes. This is a “stand-alone” flag</td>
</tr>
<tr>
<td>Constant</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PyBUF_WRITABLE</td>
<td>The returned buffer must be writable. If it is not writable, then raise an error.</td>
</tr>
<tr>
<td>PyBUF_STRIDES</td>
<td>This implies PyBUF_ND. The returned buffer must provide strides information (i.e. the strides cannot be NULL). This would be used when the consumer can handle strided, discontiguous arrays. Handling strides automatically assumes you can handle shape. The exporter can raise an error if a strided representation of the data is not possible (i.e. without the suboffsets).</td>
</tr>
<tr>
<td>PyBUF_ND</td>
<td>The returned buffer must provide shape information. The memory will be assumed C-style contiguous (last dimension varies the fastest). The exporter may raise an error if it cannot provide this kind of contiguous buffer. If this is not given then shape will be NULL.</td>
</tr>
<tr>
<td>PyBUF_C_CONTIGUOUS</td>
<td>These flags indicate that the contiguity returned buffer must be respectively, C-contiguous (last dimension varies the fastest), Fortran contiguous (first dimension varies the fastest) or either one. All of these flags imply PyBUF_STRIDES and guarantee that the strides buffer info structure will be filled in correctly.</td>
</tr>
<tr>
<td>PyBUF_F_CONTIGUOUS</td>
<td></td>
</tr>
<tr>
<td>PyBUF_ANY_CONTIGUOUS</td>
<td></td>
</tr>
<tr>
<td>PyBUF_INDIRECT</td>
<td>This flag indicates the returned buffer must have suboffsets information (which can be NULL if no suboffsets are needed). This can be used when the consumer can handle indirect array</td>
</tr>
</tbody>
</table>
referencing implied by these suboffsets. This implies **PyBUF_STRIDES**.

The returned buffer must have true format information if this flag is provided. This would be used when the consumer is going to be checking for what ‘kind’ of data is actually stored. An exporter should always be able to provide this information if requested. If format is not explicitly requested then the format must be returned as **NULL** (which means 'B', or unsigned bytes)

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyBUF_FORMAT</td>
<td>The returned buffer must have true format information if this flag is provided. An exporter should always be able to provide this information if requested. If format is not explicitly requested then the format must be returned as <strong>NULL</strong> (which means 'B', or unsigned bytes).</td>
</tr>
<tr>
<td>PyBUF_STRIDED</td>
<td>This is equivalent to ```(PyBUF_STRIDES</td>
</tr>
<tr>
<td>PyBUF_STRIDED_RO</td>
<td>This is equivalent to <code>(PyBUF_STRIDES)</code></td>
</tr>
<tr>
<td>PyBUF_RECORDS</td>
<td>This is equivalent to ```(PyBUF_STRIDES</td>
</tr>
<tr>
<td>PyBUF_RECORDS_RO</td>
<td>This is equivalent to ```(PyBUF_STRIDES</td>
</tr>
<tr>
<td>PyBUF_FULL</td>
<td>This is equivalent to ```(PyBUF_INDIRECT</td>
</tr>
<tr>
<td>PyBUF_FULL_RO</td>
<td>This is equivalent to ```(PyBUF_INDIRECT</td>
</tr>
<tr>
<td>PyBUF_CONTIG</td>
<td>This is equivalent to ```(PyBUF_ND</td>
</tr>
<tr>
<td>PyBUF_CONTIG_RO</td>
<td>This is equivalent to <code>(PyBUF_ND)</code></td>
</tr>
</tbody>
</table>

**void PyBuffer_Release(Py_buffer *view)**

Release the buffer *view*. This should be called when the buffer is no longer being used as it may free memory from it.

**Py_ssize_t PyBuffer_SizeFromFormat(const char *)**

Return the implied **itemsize** from the struct-stype **format**.

**int PyBuffer_IsContiguous(Py_buffer *view, char fortran)**
Return 1 if the memory defined by the view is C-style (\texttt{fortran} is 'C') or Fortran-style (\texttt{fortran} is 'F') contiguous or either one (\texttt{fortran} is 'A'). Return 0 otherwise.

\texttt{void PyBuffer\_FillContiguousStrides}(int \texttt{ndim},
Py\_ssize\_t \*\texttt{shape}, Py\_ssize\_t \*\texttt{strides}, Py\_ssize\_t \texttt{itemsize},
char \texttt{fortran})

Fill the \texttt{strides} array with byte-strides of a contiguous (C-style if \texttt{fortran} is 'C' or Fortran-style if \texttt{fortran} is 'F') array of the given shape with the given number of bytes per element.

\texttt{int PyBuffer\_FillInfo}(Py\_buffer \*\texttt{view}, PyObject \*\texttt{obj}, void \*\texttt{buf},
Py\_ssize\_t \texttt{len}, int \texttt{readonly}, int \texttt{infoflags})

Fill in a buffer-info structure, \texttt{view}, correctly for an exporter that can only share a contiguous chunk of memory of “unsigned bytes” of the given length. Return 0 on success and -1 (with raising an error) on error.
MemoryView objects

New in version 2.7.

A memoryview object exposes the new C level buffer interface as a Python object which can then be passed around like any other object.

PyObject *PyMemoryView_FromObject(PyObject *obj)
Create a memoryview object from an object that defines the new buffer interface.

PyObject *PyMemoryView_FromBuffer(Py_buffer *view)
Create a memoryview object wrapping the given buffer-info structure view. The memoryview object then owns the buffer, which means you shouldn’t try to release it yourself: it will be released on deallocation of the memoryview object.

PyObject *PyMemoryView_GetContiguous(PyObject *obj, int buffertype, char order)
Create a memoryview object to a contiguous chunk of memory (in either ‘C’ or ‘F’ortran order) from an object that defines the buffer interface. If memory is contiguous, the memoryview object points to the original memory. Otherwise copy is made and the memoryview points to a new bytes object.

int PyMemoryView_Check(PyObject *obj)
Return true if the object obj is a memoryview object. It is not currently allowed to create subclasses of memoryview.

Py_buffer *PyMemoryView_GET_BUFFER(PyObject *obj)
Return a pointer to the buffer-info structure wrapped by the given object. The object must be a memoryview instance; this macro doesn’t check its type, you must do it yourself or you will risk
crashes.
Old-style buffer objects

More information on the old buffer interface is provided in the section Buffer Object Structures, under the description for PyBufferProcs.

A “buffer object” is defined in the bufferobject.h header (included by Python.h). These objects look very similar to string objects at the Python programming level: they support slicing, indexing, concatenation, and some other standard string operations. However, their data can come from one of two sources: from a block of memory, or from another object which exports the buffer interface.

Buffer objects are useful as a way to expose the data from another object’s buffer interface to the Python programmer. They can also be used as a zero-copy slicing mechanism. Using their ability to reference a block of memory, it is possible to expose any data to the Python programmer quite easily. The memory could be a large, constant array in a C extension, it could be a raw block of memory for manipulation before passing to an operating system library, or it could be used to pass around structured data in its native, in-memory format.

PyBufferObject
This subtype of PyObject represents a buffer object.

PyTypeObject PyBuffer_Type
The instance of PyTypeObject which represents the Python buffer type; it is the same object as buffer and types.BufferType in the Python layer.

int Py_END_OF_BUFFER
This constant may be passed as the size parameter to PyBuffer_FromObject() or PyBuffer_FromReadWriteObject(). It
indicates that the new `PyBufferObject` should refer to `base` object from the specified `offset` to the end of its exported buffer. Using this enables the caller to avoid querying the `base` object for its length.

```c
int PyBuffer_Check(PyObject *p)
Return true if the argument has type `PyBuffer_Type`.
```

```c
PyObject* PyBuffer_FromObject(PyObject *base,
Py_ssize_t offset, Py_ssize_t size)
Return a new read-only buffer object. This raises `TypeError` if base doesn't support the read-only buffer protocol or doesn't provide exactly one buffer segment, or it raises `ValueError` if offset is less than zero. The buffer will hold a reference to the base object, and the buffer's contents will refer to the base object's buffer interface, starting as position offset and extending for size bytes. If size is `Py_END_OF_BUFFER`, then the new buffer's contents extend to the length of the base object's exported buffer data.

Changed in version 2.5: This function used an `int` type for offset and size. This might require changes in your code for properly supporting 64-bit systems.
```

```c
PyObject* PyBuffer_FromReadWriteObject(PyObject *base,
Py_ssize_t offset, Py_ssize_t size)
Return a new writable buffer object. Parameters and exceptions are similar to those for `PyBuffer_FromObject()`. If the base object does not export the writeable buffer protocol, then `TypeError` is raised.

Changed in version 2.5: This function used an `int` type for offset
and size. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyBuffer_FromMemory(void *ptr, Py_ssize_t size)

*Return value: New reference.*
Return a new read-only buffer object that reads from a specified location in memory, with a specified size. The caller is responsible for ensuring that the memory buffer, passed in as ptr, is not deallocated while the returned buffer object exists. Raises ValueError if size is less than zero. Note that Py_END_OF_BUFFER may not be passed for the size parameter; ValueError will be raised in that case.

*Changed in version 2.5:* This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyBuffer_FromReadWriteMemory(void *ptr, Py_ssize_t size)

*Return value: New reference.*
Similar to PyBuffer_FromMemory(), but the returned buffer is writable.

*Changed in version 2.5:* This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyBuffer_New(Py_ssize_t size)

*Return value: New reference.*
Return a new writable buffer object that maintains its own memory buffer of size bytes. ValueError is returned if size is not zero or positive. Note that the memory buffer (as returned by PyObject_AsWriteBuffer()) is not specifically aligned.

*Changed in version 2.5:* This function used an int type for size.
This might require changes in your code for properly supporting 64-bit systems.
Tuple Objects

**PyTupleObject**
This subtype of **PyObject** represents a Python tuple object.

**PyTypeObject PyTuple_Type**
This instance of **PyTypeObject** represents the Python tuple type; it is the same object as `tuple` and `types.TupleType` in the Python layer.

```c
int PyTuple_Check(PyObject *p)
```
Return true if `p` is a tuple object or an instance of a subtype of the tuple type.

*Changed in version 2.2:* Allowed subtypes to be accepted.

```c
int PyTuple_CheckExact(PyObject *p)
```
Return true if `p` is a tuple object, but not an instance of a subtype of the tuple type.

*New in version 2.2.*

```c
PyObject* PyTuple_New(Py_ssize_t len)
```
*Return value: New reference.*
Return a new tuple object of size `len`, or `NULL` on failure.

*Changed in version 2.5:* This function used an `int` type for `len`. This might require changes in your code for properly supporting 64-bit systems.

```c
PyObject* PyTuple_Pack(Py_ssize_t n, ...)
```
*Return value: New reference.*
Return a new tuple object of size `n`, or `NULL` on failure. The tuple values are initialized to the subsequent `n` C arguments pointing to
Python objects.  PyTuple_Pack(2, a, b) is equivalent to Py_BuildValue("(OO)", a, b).

New in version 2.4.

Changed in version 2.5: This function used an int type for n. This might require changes in your code for properly supporting 64-bit systems.

Py_ssize_t PyTuple_Size(PyObject *p)

Take a pointer to a tuple object, and return the size of that tuple.

Changed in version 2.5: This function returned an int type. This might require changes in your code for properly supporting 64-bit systems.

Py_ssize_t PyTuple_GET_SIZE(PyObject *p)

Return the size of the tuple p, which must be non-NULL and point to a tuple; no error checking is performed.

Changed in version 2.5: This function returned an int type. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyTuple_GetItem(PyObject *p, Py_ssize_t pos)

Return value: Borrowed reference.

Return the object at position pos in the tuple pointed to by p. If pos is out of bounds, return NULL and sets an IndexError exception.

Changed in version 2.5: This function used an int type for pos. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyTuple_GET_ITEM(PyObject *p, Py_ssize_t pos)
Return value: Borrowed reference.
Like PyTuple_GetItem(), but does no checking of its arguments.

Changed in version 2.5: This function used an int type for pos. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyTuple_GetSlice(PyObject *p, Py_ssize_t low, Py_ssize_t high)
Take a slice of the tuple pointed to by p from low to high and return it as a new tuple.

Changed in version 2.5: This function used an int type for low and high. This might require changes in your code for properly supporting 64-bit systems.

int PyTuple_SetItem(PyObject *p, Py_ssize_t pos, PyObject *o)
Insert a reference to object o at position pos of the tuple pointed to by p. Return 0 on success.

Note: This function “steals” a reference to o.

Changed in version 2.5: This function used an int type for pos. This might require changes in your code for properly supporting 64-bit systems.

void PyTuple_SET_ITEM(PyObject *p, Py_ssize_t pos, PyObject *o)
Like PyTuple_SetItem(), but does no error checking, and should only be used to fill in brand new tuples.

Note: This function “steals” a reference to o.

Changed in version 2.5: This function used an int type for pos. This might require changes in your code for properly supporting
64-bit systems.

```c
int _PyTuple_Resize(PyObject **p, Py_ssize_t newsize)
```

Can be used to resize a tuple. `newsize` will be the new length of the tuple. Because tuples are supposed to be immutable, this should only be used if there is only one reference to the object. Do not use this if the tuple may already be known to some other part of the code. The tuple will always grow or shrink at the end. Think of this as destroying the old tuple and creating a new one, only more efficiently. Returns 0 on success. Client code should never assume that the resulting value of `p` will be the same as before calling this function. If the object referenced by `p` is replaced, the original `p` is destroyed. On failure, returns -1 and sets `p` to `NULL`, and raises `MemoryError` or `SystemError`.

*Changed in version 2.2:* Removed unused third parameter, `last_is_sticky`.

*Changed in version 2.5:* This function used an `int` type for `newsize`. This might require changes in your code for properly supporting 64-bit systems.

```c
int PyTuple_ClearFreeList()
```

Clear the free list. Return the total number of freed items.

*New in version 2.6.*
**List Objects**

**PyListObject**
This subtype of `PyObject` represents a Python list object.

**PyTypeObject PyList_Type**
This instance of `PyTypeObject` represents the Python list type. This is the same object as `list` in the Python layer.

```c
int PyList_Check(PyObject *p)
```

Return true if `p` is a list object or an instance of a subtype of the list type.

*Changed in version 2.2:* Allowed subtypes to be accepted.

```c
int PyList_CheckExact(PyObject *p)
```

Return true if `p` is a list object, but not an instance of a subtype of the list type.

*New in version 2.2.*

```c
PyObject* PyList_New(Py_ssize_t len)
```

*Return value: New reference.*

Return a new list of length `len` on success, or `NULL` on failure.

**Note:** If `len` is greater than zero, the returned list object’s items are set to `NULL`. Thus you cannot use abstract API functions such as `PySequence_SetItem()` or expose the object to Python code before setting all items to a real object with `PyList_SetItem()`.

*Changed in version 2.5:* This function used an `int` for `size`. This might require changes in your code for properly supporting 64-bit systems.
Py_ssize_t PyList_Size(PyObject *list)

Return the length of the list object in list; this is equivalent to len(list) on a list object.

Changed in version 2.5: This function returned an int. This might require changes in your code for properly supporting 64-bit systems.

Py_ssize_t PyList_GET_SIZE(PyObject *list)

Macro form of PyList_Size() without error checking.

Changed in version 2.5: This macro returned an int. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyList_GetItem(PyObject *list, Py_ssize_t index)

Return value: Borrowed reference.
Return the object at position index in the list pointed to by list. The position must be positive, indexing from the end of the list is not supported. If index is out of bounds, return NULL and set an IndexError exception.

Changed in version 2.5: This function used an int for index. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyList_GET_ITEM(PyObject *list, Py_ssize_t i)

Return value: Borrowed reference.
Macro form of PyList_GetItem() without error checking.

Changed in version 2.5: This macro used an int for i. This might require changes in your code for properly supporting 64-bit systems.

int PyList_SetItem(PyObject *list, Py_ssize_t index,
PyObject *item)

Set the item at index index in list to item. Return 0 on success or -1 on failure.

**Note:** This function “steals” a reference to item and discards a reference to an item already in the list at the affected position.

*Changed in version 2.5:* This function used an int for index. This might require changes in your code for properly supporting 64-bit systems.

```c
void PyList_SET_ITEM(PyObject *list, Py_ssize_t i, PyObject *o)
```

Macro form of PyList_SetItem() without error checking. This is normally only used to fill in new lists where there is no previous content.

**Note:** This macro “steals” a reference to item, and, unlike PyList_SetItem(), does not discard a reference to any item that it being replaced; any reference in list at position i will be leaked.

*Changed in version 2.5:* This macro used an int for i. This might require changes in your code for properly supporting 64-bit systems.

```c
int PyList_Insert(PyObject *list, Py_ssize_t index, PyObject *item)
```

Insert the item item into list list in front of index index. Return 0 if successful; return -1 and set an exception if unsuccessful. Analogous to list.insert(index, item).

*Changed in version 2.5:* This function used an int for index. This might require changes in your code for properly supporting 64-bit systems.
int **PyList_Append** (**PyObject** *list, **PyObject** *item*)

Append the object *item* at the end of list *list*. Return 0 if successful; return -1 and set an exception if unsuccessful. Analogous to `list.append(item)`.

**PyObject** *PyList_GetSlice** (**PyObject** *list, Py_ssize_t *low, Py_ssize_t *high*)

*Return value: New reference.*

Return a list of the objects in *list* containing the objects between *low* and *high*. Return NULL and set an exception if unsuccessful. Analogous to `list[low:high]`. Negative indices, as when slicing from Python, are not supported.

*Changed in version 2.5:* This function used an int for *low* and *high*. This might require changes in your code for properly supporting 64-bit systems.

int **PyList_SetSlice** (**PyObject** *list, Py_ssize_t *low, Py_ssize_t *high, **PyObject** *itemlist*)

Set the slice of *list* between *low* and *high* to the contents of *itemlist*. Analogous to `list[low:high] = itemlist`. The *itemlist* may be NULL, indicating the assignment of an empty list (slice deletion). Return 0 on success, -1 on failure. Negative indices, as when slicing from Python, are not supported.

*Changed in version 2.5:* This function used an int for *low* and *high*. This might require changes in your code for properly supporting 64-bit systems.

int **PyList_Sort** (**PyObject** *list*)

Sort the items of *list* in place. Return 0 on success, -1 on failure. This is equivalent to `list.sort()`.

int **PyList_Reverse** (**PyObject** *list*)
Reverse the items of list in place. Return 0 on success, -1 on failure. This is the equivalent of \texttt{list.reverse()}. 

\texttt{PyObject* PyList_AsTuple(PyObject *list)}

\textit{Return value: New reference.}

Return a new tuple object containing the contents of \texttt{list}; equivalent to \texttt{tuple(list)}.
Dictionary Objects

**PyDictObject**
This subtype of **PyObject** represents a Python dictionary object.

**PyTypeObject PyDict_Type**
This instance of **PyTypeObject** represents the Python dictionary type. This is exposed to Python programs as `dict` and `types.DictType`.

```c
int PyDict_Check(PyObject *p)
```
Return true if `p` is a dict object or an instance of a subtype of the dict type.

*Changed in version 2.2:* Allowed subtypes to be accepted.

```c
int PyDict_CheckExact(PyObject *p)
```
Return true if `p` is a dict object, but not an instance of a subtype of the dict type.

*New in version 2.4.*

```c
PyObject* PyDict_New()
```
*Return value: New reference.*
Return a new empty dictionary, or `NULL` on failure.

```c
PyObject* PyDictProxy_New(PyObject *dict)
```
*Return value: New reference.*
Return a proxy object for a mapping which enforces read-only behavior. This is normally used to create a proxy to prevent modification of the dictionary for non-dynamic class types.

*New in version 2.2.*

```c
void PyDict_Clear(PyObject *p)
```
Empty an existing dictionary of all key-value pairs.

int PyDict_Contains(PyObject *p, PyObject *key)
Determine if dictionary p contains key. If an item in p is matches key, return 1, otherwise return 0. On error, return -1. This is equivalent to the Python expression key in p.

New in version 2.4.

PyObject* PyDict_Copy(PyObject *p)
Return a new dictionary that contains the same key-value pairs as p.

New in version 1.6.

int PyDict_SetItem(PyObject *p, PyObject *key, PyObject *val)
Insert value into the dictionary p with a key of key. key must be hashable; if it isn’t, TypeError will be raised. Return 0 on success or -1 on failure.

int PyDict_SetItemString(PyObject *p, const char *key, PyObject *val)
Insert value into the dictionary p using key as a key. key should be a char*. The key object is created using PyString_FromString(key). Return 0 on success or -1 on failure.

int PyDict_DelItem(PyObject *p, PyObject *key)
Remove the entry in dictionary p with key key. key must be hashable; if it isn’t, TypeError is raised. Return 0 on success or -1 on failure.

int PyDict_DelItemString(PyObject *p, char *key)
Remove the entry in dictionary p which has a key specified by the string key. Return 0 on success or -1 on failure.
PyObject* PyDict_GetItem(PyObject *p, PyObject *key)

*Return value: Borrowed reference.*

Return the object from dictionary p which has a key key. Return NULL if the key key is not present, but without setting an exception.

PyObject* PyDict_GetItemString(PyObject *p, const char *key)

*Return value: Borrowed reference.*

This is the same as PyDict_GetItem(), but key is specified as a char*, rather than a PyObject*.

PyObject* PyDict_Items(PyObject *p)

*Return value: New reference.*

Return a PyListObject containing all the items from the dictionary, as in the dictionary method dict.items().

PyObject* PyDict_Keys(PyObject *p)

*Return value: New reference.*

Return a PyListObject containing all the keys from the dictionary, as in the dictionary method dict.keys().

PyObject* PyDict_Values(PyObject *p)

*Return value: New reference.*

Return a PyListObject containing all the values from the dictionary p, as in the dictionary method dict.values().

Py_ssize_t PyDict_Size(PyObject *p)

Return the number of items in the dictionary. This is equivalent to len(p) on a dictionary.

Changed in version 2.5: This function returned an int type. This might require changes in your code for properly supporting 64-bit systems.
int PyDict_Next(PyObject *p, Py_ssize_t *ppos, PyObject **pkey, PyObject **pvalue)

Iterate over all key-value pairs in the dictionary `p`. The `Py_ssize_t` referred to by `ppos` must be initialized to 0 prior to the first call to this function to start the iteration; the function returns true for each pair in the dictionary, and false once all pairs have been reported. The parameters `pkey` and `pvalue` should either point to `PyObject*` variables that will be filled in with each key and value, respectively, or may be `NULL`. Any references returned through them are borrowed. `ppos` should not be altered during iteration. Its value represents offsets within the internal dictionary structure, and since the structure is sparse, the offsets are not consecutive.

For example:

```c
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* do something interesting with the values... */
    ...
}
```

The dictionary `p` should not be mutated during iteration. It is safe (since Python 2.1) to modify the values of the keys as you iterate over the dictionary, but only so long as the set of keys does not change. For example:

```c
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    int i = PyInt_AS_LONG(value) + 1;
    PyObject *o = PyInt_FromLong(i);
    if (o == NULL)
        return -1;
    if (PyDict_SetItem(self->dict, key, o) < 0) {
        Py_DECREF(o);
        return -1;
    }
```
Changed in version 2.5: This function used an int * type for ppos. This might require changes in your code for properly supporting 64-bit systems.

int PyDict_Merge(PyObject *a, PyObject *b, int override)
Iterate over mapping object b adding key-value pairs to dictionary a. b may be a dictionary, or any object supporting PyMapping_Keys() and PyObject_GetItem(). If override is true, existing pairs in a will be replaced if a matching key is found in b, otherwise pairs will only be added if there is not a matching key in a. Return 0 on success or -1 if an exception was raised.

New in version 2.2.

int PyDict_Update(PyObject *a, PyObject *b)
This is the same as PyDict_Merge(a, b, 1) in C, and is similar to a.update(b) in Python except that PyDict_Update() doesn’t fall back to the iterating over a sequence of key value pairs if the second argument has no “keys” attribute. Return 0 on success or -1 if an exception was raised.

New in version 2.2.

int PyDict_MergeFromSeq2(PyObject *a, PyObject *seq2, int override)
Update or merge into dictionary a, from the key-value pairs in seq2. seq2 must be an iterable object producing iterable objects of length 2, viewed as key-value pairs. In case of duplicate keys, the last wins if override is true, else the first wins. Return 0 on success or -1 if an exception was raised. Equivalent Python (except for the return value):
def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
        if override or key not in a:
            a[key] = value

New in version 2.2.
Class and Instance Objects

Note that the class objects described here represent old-style classes, which will go away in Python 3. When creating new types for extension modules, you will want to work with type objects (section Type Objects).

**PyClassObject**

The C structure of the objects used to describe built-in classes.

**PyObject**\* **PyClass_Type**

This is the type object for class objects; it is the same object as \texttt{types.ClassType} in the Python layer.

\begin{verbatim}
int PyClass_Check(PyObject \*o)
    Return true if the object \(o\) is a class object, including instances of types derived from the standard class object. Return false in all other cases.
\end{verbatim}

\begin{verbatim}
int PyClass_IsSubclass(PyObject \*klass, PyObject \*base)
    Return true if \(\texttt{klass}\) is a subclass of \(\texttt{base}\). Return false in all other cases.
\end{verbatim}

There are very few functions specific to instance objects.

**PyTypeObject** **PyInstance_Type**

Type object for class instances.

\begin{verbatim}
int PyInstance_Check(PyObject \*obj)
    Return true if \(\texttt{obj}\) is an instance.
\end{verbatim}

\begin{verbatim}
PyObject\* PyInstance_New(PyObject \*class, PyObject \*arg, PyObject \*kw)
    Create a new instance of a specific class. The parameters \texttt{arg}
and \( kw \) are used as the positional and keyword parameters to the object's constructor.

\[
\text{PyObject* } \text{PyInstance\_NewRaw}(\text{PyObject* class, PyObject* dict})
\]

*Return value: New reference.*

Create a new instance of a specific class without calling its constructor. `class` is the class of new object. The `dict` parameter will be used as the object's `__dict__`; if `NULL`, a new dictionary will be created for the instance.
Function Objects

There are a few functions specific to Python functions.

**PyFunctionObject**

The C structure used for functions.

**PyTypeObject PyFunction_Type**

This is an instance of PyTypeObject and represents the Python function type. It is exposed to Python programmers as types.FunctionType.

```c
int PyFunction_Check(PyObject *o)
```

Return true if o is a function object (has type PyFunction_Type). The parameter must not be NULL.

```c
PyObject* PyFunction_New(PyObject *code, PyObject *globals)
```

*Return value: New reference.*

Return a new function object associated with the code object code. globals must be a dictionary with the global variables accessible to the function.

The function's docstring, name and __module__ are retrieved from the code object, the argument defaults and closure are set to NULL.

```c
PyObject* PyFunction_GetCode(PyObject *op)
```

*Return value: Borrowed reference.*

Return the code object associated with the function object op.

```c
PyObject* PyFunction_GetGlobals(PyObject *op)
```

*Return value: Borrowed reference.*

Return the globals dictionary associated with the function object op.
PyObject* PyFunction_GetModule(PyObject *op)

Return value: Borrowed reference.
Return the __module__ attribute of the function object op. This is normally a string containing the module name, but can be set to any other object by Python code.

PyObject* PyFunction_GetDefaults(PyObject *op)

Return value: Borrowed reference.
Return the argument default values of the function object op. This can be a tuple of arguments or NULL.

int PyFunction_SetDefaults(PyObject *op, PyObject *defaults)

Set the argument default values for the function object op. defaults must be Py_None or a tuple.

Raises SystemError and returns -1 on failure.

PyObject* PyFunction_GetClosure(PyObject *op)

Return value: Borrowed reference.
Return the closure associated with the function object op. This can be NULL or a tuple of cell objects.

int PyFunction_SetClosure(PyObject *op, PyObject *closure)

Set the closure associated with the function object op. closure must be Py_None or a tuple of cell objects.

Raises SystemError and returns -1 on failure.
There are some useful functions that are useful for working with method objects.

**PyTypeObject PyMethod_Type**

This instance of PyTypeObject represents the Python method type. This is exposed to Python programs as types.MethodType.

`int PyMethod_Check(PyObject *o)`

Return true if `o` is a method object (has type PyMethod_Type). The parameter must not be NULL.

`PyObject* PyMethod_New(PyObject *func, PyObject *self, PyObject *class)`

*Return value: New reference.*

Return a new method object, with `func` being any callable object; this is the function that will be called when the method is called. If this method should be bound to an instance, `self` should be the instance and `class` should be the class of `self`, otherwise `self` should be NULL and `class` should be the class which provides the unbound method..

`PyObject* PyMethod_Class(PyObject *meth)`

*Return value: Borrowed reference.*

Return the class object from which the method `meth` was created; if this was created from an instance, it will be the class of the instance.

`PyObject* PyMethod_GET_CLASS(PyObject *meth)`

*Return value: Borrowed reference.*

Macro version of PyMethod_Class() which avoids error checking.

`PyObject* PyMethod_Function(PyObject *meth)`
Return value: Borrowed reference.
Return the function object associated with the method \texttt{meth}.

\texttt{PyObject* PyMethod\_GET\_FUNCTION(PyObject *meth)}

Return value: Borrowed reference.
Macro version of \texttt{PyMethod\_Function()} which avoids error checking.

\texttt{PyObject* PyMethod\_Self(PyObject *meth)}

Return value: Borrowed reference.
Return the instance associated with the method \texttt{meth} if it is bound, otherwise return \texttt{NULL}.

\texttt{PyObject* PyMethod\_GET\_SELF(PyObject *meth)}

Return value: Borrowed reference.
Macro version of \texttt{PyMethod\_Self()} which avoids error checking.

\texttt{int PyMethod\_Clear\_Free\_List()} 

Clear the free list. Return the total number of freed items.

New in version 2.6.
File Objects

Python's built-in file objects are implemented entirely on the FILE* support from the C standard library. This is an implementation detail and may change in future releases of Python.

**PyFileObject**

This subtype of PyObject represents a Python file object.

**PyTypeObject PyFile_Type**

This instance of PyTypeObject represents the Python file type. This is exposed to Python programs as file and types.FileType.

```c
int PyFile_Check(PyObject *p)
```

Return true if its argument is a PyFileObject or a subtype of PyFileObject.

*Changed in version 2.2:* Allowed subtypes to be accepted.

```c
int PyFile_CheckExact(PyObject *p)
```

Return true if its argument is a PyFileObject, but not a subtype of PyFileObject.

*New in version 2.2.*

```c
PyObject* PyFile_FromString(char *filename, char *mode)
```

*Return value: New reference.*

On success, return a new file object that is opened on the file given by filename, with a file mode given by mode, where mode has the same semantics as the standard C routine fopen(). On failure, return NULL.

```c
PyObject* PyFile_FromFile(FILE *fp, char *name, char *mode, int (*close)(FILE*))
```


**Return value: New reference.**
Create a new `PyFileObject` from the already-open standard C file pointer, `fp`. The function `close` will be called when the file should be closed. Return `NULL` and close the file using `close` on failure. `close` is optional and can be set to `NULL`.

```c
FILE* PyFile_AsFile(PyObject *p)
```
Return the file object associated with `p` as a `FILE*`.

If the caller will ever use the returned `FILE*` object while the GIL is released it must also call the `PyFile_IncUseCount()` and `PyFile_DecUseCount()` functions described below as appropriate.

```c
void PyFile_IncUseCount(PyFileObject *p)
```
Increments the PyFileObject’s internal use count to indicate that the underlying `FILE*` is being used. This prevents Python from calling `f_close()` on it from another thread. Callers of this must call `PyFile_DecUseCount()` when they are finished with the `FILE*`. Otherwise the file object will never be closed by Python.

The GIL must be held while calling this function.

The suggested use is to call this after `PyFile_AsFile()` and before you release the GIL:

```c
FILE *fp = PyFile_AsFile(p);
PyFile_IncUseCount(p);
/* ... */
Py_BEGIN_ALLOW_THREADS
do_something(fp);
Py_END_ALLOW_THREADS
/* ... */
PyFile_DecUseCount(p);
```

*New in version 2.6.*

```c
void PyFile_DecUseCount(PyFileObject *p)
```
Decrement the PyFileObject’s internal unlocked_count member to indicate that the caller is done with its own use of the FILE*. This may only be called to undo a prior call to PyFile_IncUseCount().

The GIL must be held while calling this function (see the example above).

New in version 2.6.

PyObject* PyFile_GetLine(PyObject *p, int n)

Equivalent to p.readline(n), this function reads one line from the object p. p may be a file object or any object with a readline() method. If n is 0, exactly one line is read, regardless of the length of the line. If n is greater than 0, no more than n bytes will be read from the file; a partial line can be returned. In both cases, an empty string is returned if the end of the file is reached immediately. If n is less than 0, however, one line is read regardless of length, but EOFError is raised if the end of the file is reached immediately.

PyObject* PyFile_Name(PyObject *p)

Return value: Borrowed reference.
Return the name of the file specified by p as a string object.

void PyFile_SetBufSize(PyFileObject *p, int n)
Available on systems with setvbuf() only. This should only be called immediately after file object creation.

int PyFile_SetEncoding(PyFileObject *p, const char *enc)
Set the file’s encoding for Unicode output to enc. Return 1 on success and 0 on failure.
New in version 2.3.

```c
int PyFile_SetEncodingAndErrors(PyFileObject *p, const char *enc, *errors)
```

Set the file’s encoding for Unicode output to `enc`, and its error mode to `err`. Return 1 on success and 0 on failure.

New in version 2.6.

```c
int PyFile_SoftSpace(PyObject *p, int newflag)
```

This function exists for internal use by the interpreter. Set the `softspace` attribute of `p` to `newflag` and return the previous value. `p` does not have to be a file object for this function to work properly; any object is supported (though its only interesting if the `softspace` attribute can be set). This function clears any errors, and will return 0 as the previous value if the attribute either does not exist or if there were errors in retrieving it. There is no way to detect errors from this function, but doing so should not be needed.

```c
int PyFile_WriteObject(PyObject *obj, PyObject *p, int flags)
```

Write object `obj` to file object `p`. The only supported flag for `flags` is `Py_PRINT_RAW`; if given, the `str()` of the object is written instead of the `repr()`. Return 0 on success or -1 on failure; the appropriate exception will be set.

```c
int PyFile_WriteString(const char *s, PyObject *p)
```

Write string `s` to file object `p`. Return 0 on success or -1 on failure; the appropriate exception will be set.
Module Objects

There are only a few functions special to module objects.

** PyTypeObject PyModule_Type
   This instance of PyTypeObject represents the Python module type. This is exposed to Python programs as types.ModuleType.

int PyModule_Check(PyObject *p)
   Return true if p is a module object, or a subtype of a module object.

   *Changed in version 2.2:* Allowed subtypes to be accepted.

int PyModule_CheckExact(PyObject *p)
   Return true if p is a module object, but not a subtype of PyModule_Type.

   *New in version 2.2.*

PyObject* PyModule_New(const char *name)
   *Return value: New reference.*
   Return a new module object with the __name__ attribute set to name. Only the module’s __doc__ and __name__ attributes are filled in; the caller is responsible for providing a __file__ attribute.

PyObject* PyModule_GetDict(PyObject *module)
   *Return value: Borrowed reference.*
   Return the dictionary object that implements module’s namespace; this object is the same as the __dict__ attribute of the module object. This function never fails. It is recommended extensions use other PyModule_*() and PyObject_*() functions rather than directly manipulate a module’s __dict__.
char* PyModule_GetName(PyObject *module)
    Return module's __name__ value. If the module does not provide one, or if it is not a string, SystemError is raised and NULL is returned.

cchar* PyModule_GetFilename(PyObject *module)
    Return the name of the file from which module was loaded using module's __file__ attribute. If this is not defined, or if it is not a string, raise SystemError and return NULL.

int PyModule_AddObject(PyObject *module, const char *name, PyObject *value)
    Add an object to module as name. This is a convenience function which can be used from the module's initialization function. This steals a reference to value. Return -1 on error, 0 on success.

    New in version 2.0.

int PyModule_AddIntConstant(PyObject *module, const char *name, long value)
    Add an integer constant to module as name. This convenience function can be used from the module's initialization function. Return -1 on error, 0 on success.

    New in version 2.0.

int PyModule_AddStringConstant(PyObject *module, const char *name, const char *value)
    Add a string constant to module as name. This convenience function can be used from the module's initialization function. The string value must be null-terminated. Return -1 on error, 0 on success.

    New in version 2.0.
**int PyModule_AddIntMacro(**PyObject *module, macro)**

Add an int constant to *module*. The name and the value are taken from *macro*. For example `PyModule_AddIntMacro(module, AF_INET)` adds the int constant `AF_INET` with the value of `AF_INET` to *module*. Return `-1` on error, `0` on success.

*New in version 2.6.*

**int PyModule_AddStringMacro(**PyObject *module, macro)**

Add a string constant to *module*.

*New in version 2.6.*
Python provides two general-purpose iterator objects. The first, a sequence iterator, works with an arbitrary sequence supporting the \_\_getitem\_\_() method. The second works with a callable object and a sentinel value, calling the callable for each item in the sequence, and ending the iteration when the sentinel value is returned.

**PyTypeObject** **PySeqIter\_Type**
Type object for iterator objects returned by **PySeqIter\_New()** and the one-argument form of the **iter()** built-in function for built-in sequence types.

*New in version 2.2.*

**int** **PySeqIter\_Check**(op)
Return true if the type of op is **PySeqIter\_Type**.

*New in version 2.2.*

**PyObject**\* **PySeqIter\_New**(PyObject \*seq)
*Return value: New reference.*
Return an iterator that works with a general sequence object, seq. The iteration ends when the sequence raises **IndexError** for the subscripting operation.

*New in version 2.2.*

**PyTypeObject** **PyCallIter\_Type**
Type object for iterator objects returned by **PyCallIter\_New()** and the two-argument form of the **iter()** built-in function.

*New in version 2.2.*
int **PyCallIter_Check**(op)

Return true if the type of op is **PyCallIter_Type**.

*New in version 2.2.*

**PyObject** * **PyCallIter_New**(PyObject **callable**, PyObject **sentinel**)

*Return value: New reference.*

Return a new iterator. The first parameter, **callable**, can be any Python callable object that can be called with no parameters; each call to it should return the next item in the iteration. When **callable** returns a value equal to **sentinel**, the iteration will be terminated.

*New in version 2.2.*
Descriptor Objects

“Descriptors” are objects that describe some attribute of an object. They are found in the dictionary of type objects.

**PyTypeObject** **PyProperty_Type**

The type object for the built-in descriptor types.

*New in version 2.2.*

**PyObject**

*PyDescr_NewGetSet*(PyTypeObject *type, struct PyGetSetDef *getset)

*Return value: New reference.*

*New in version 2.2.*

**PyObject**

*PyDescr_NewMember*(PyTypeObject *type, struct PyMemberDef *meth)

*Return value: New reference.*

*New in version 2.2.*

**PyObject**

*PyDescr_NewMethod*(PyTypeObject *type, struct PyMethodDef *meth)

*Return value: New reference.*

*New in version 2.2.*

**PyObject**

*PyDescr_NewWrapper*(PyTypeObject *type, struct wrapperbase *wrapper, void *wrapped)

*Return value: New reference.*

*New in version 2.2.*

**PyObject**

*PyDescr_NewClassMethod*(PyTypeObject *type, PyMethodDef *method)

*Return value: New reference.*

*New in version 2.3.*

**int** **PyDescr_IsData**(PyObject *descr)
Return true if the descriptor objects `descr` describes a data attribute, or false if it describes a method. `descr` must be a descriptor object; there is no error checking.

*New in version 2.2.*

```c
PyObject* PyWrapper_New(PyObject *, PyObject *)
```

*Return value: New reference.*

*New in version 2.2.*
Slice Objects

PyObject PySlice_Type
The type object for slice objects. This is the same as slice and types.SliceType.

int PySlice_Check(PyObject *ob)
Return true if ob is a slice object; ob must not be NULL.

PyObject* PySlice_New(PyObject *start, PyObject *stop, PyObject *step)
Return a new slice object with the given values. The start, stop, and step parameters are used as the values of the slice object attributes of the same names. Any of the values may be NULL, in which case the None will be used for the corresponding attribute. Return NULL if the new object could not be allocated.

int PySlice_GetIndices(PySliceObject *slice, Py_ssize_t length, Py_ssize_t *start, Py_ssize_t *stop, Py_ssize_t *step)
Retrieve the start, stop and step indices from the slice object slice, assuming a sequence of length length. Treats indices greater than length as errors.

Returns 0 on success and -1 on error with no exception set (unless one of the indices was not None and failed to be converted to an integer, in which case -1 is returned with an exception set).

You probably do not want to use this function. If you want to use slice objects in versions of Python prior to 2.3, you would probably do well to incorporate the source of PySlice_GetIndicesEx(), suitably renamed, in the source of your extension.
Changed in version 2.5: This function used an `int` type for `length` and an `int *` type for `start`, `stop`, and `step`. This might require changes in your code for properly supporting 64-bit systems.

```c
int PySlice_GetIndicesEx(PySliceObject *slice, Py_ssize_t length, Py_ssize_t *start, Py_ssize_t *stop, Py_ssize_t *step, Py_ssize_t *slicelength)
```

Usable replacement for `PySlice_GetIndices()`. Retrieve the start, stop, and step indices from the slice object `slice` assuming a sequence of length `length`, and store the length of the slice in `slicelength`. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Returns 0 on success and -1 on error with exception set.

New in version 2.3.

Changed in version 2.5: This function used an `int` type for `length` and an `int *` type for `start`, `stop`, `step`, and `slicelength`. This might require changes in your code for properly supporting 64-bit systems.
Weak Reference Objects

Python supports *weak references* as first-class objects. There are two specific object types which directly implement weak references. The first is a simple reference object, and the second acts as a proxy for the original object as much as it can.

```c
int PyWeakref_Check(ob)
    Return true if *ob* is either a reference or proxy object.

    *New in version 2.2.*
```

```c
int PyWeakref_CheckRef(ob)
    Return true if *ob* is a reference object.

    *New in version 2.2.*
```

```c
int PyWeakref_CheckProxy(ob)
    Return true if *ob* is a proxy object.

    *New in version 2.2.*
```

```c
PyObject* PyWeakref_NewRef(PyObject *ob, PyObject *callback)

    *Return value: New reference.*
    Return a weak reference object for the object *ob*. This will always return a new reference, but is not guaranteed to create a new object; an existing reference object may be returned. The second parameter, *callback*, can be a callable object that receives notification when *ob* is garbage collected; it should accept a single parameter, which will be the weak reference object itself. *callback* may also be None or NULL. If *ob* is not a weakly-referencable object, or if *callback* is not callable, None, or NULL, this will return NULL and raise TypeError.

    *New in version 2.2.*
```
PyObject* PyWeakref_NewProxy(PyObject *ob, PyObject *callback)

*Return value: New reference.*

Return a weak reference proxy object for the object `ob`. This will always return a new reference, but is not guaranteed to create a new object; an existing proxy object may be returned. The second parameter, `callback`, can be a callable object that receives notification when `ob` is garbage collected; it should accept a single parameter, which will be the weak reference object itself. `callback` may also be `None` or `NULL`. If `ob` is not a weakly-referencable object, or if `callback` is not callable, `None`, or `NULL`, this will return `NULL` and raise `TypeError`.

*New in version 2.2.*

PyObject* PyWeakref_GetObject(PyObject *ref)

*Return value: Borrowed reference.*

Return the referenced object from a weak reference, `ref`. If the referent is no longer live, returns `Py_None`.

*New in version 2.2.*

**Warning:** This function returns a borrowed reference to the referenced object. This means that you should always call `Py_INCREF()` on the object except if you know that it cannot be destroyed while you are still using it.

PyObject* PyWeakref_GET_OBJECT(PyObject *ref)

*Return value: Borrowed reference.*

Similar to `PyWeakref_GetObject()`, but implemented as a macro that does no error checking.

*New in version 2.2.*
Capsules

Refer to Providing a C API for an Extension Module for more information on using these objects.

PyCapsule

This subtype of PyObject represents an opaque value, useful for C extension modules who need to pass an opaque value (as a void* pointer) through Python code to other C code. It is often used to make a C function pointer defined in one module available to other modules, so the regular import mechanism can be used to access C APIs defined in dynamically loaded modules.

PyCapsule_Destructor

The type of a destructor callback for a capsule. Defined as:

```c
typedef void (*PyCapsule_Destructor)(PyObject *);
```

See PyCapsule_New() for the semantics of PyCapsule_Destructor callbacks.

int PyCapsule_CheckExact(PyObject *p)

Return true if its argument is a PyCapsule.

PyObject* PyCapsule_New(void *pointer, const char *name, PyCapsule_Destructor destructor)

*Return value: New reference.*

Create a PyCapsule encapsulating the pointer. The pointer argument may not be NULL.

On failure, set an exception and return NULL.

The name string may either be NULL or a pointer to a valid C string. If non-NULL, this string must outlive the capsule. (Though
it is permitted to free it inside the destructor.

If the destructor argument is not NULL, it will be called with the capsule as its argument when it is destroyed.

If this capsule will be stored as an attribute of a module, the name should be specified as `modulename.attributename`. This will enable other modules to import the capsule using `PyCapsule.Import()`.

```c
void* PyCapsule_GetPointer(PyObject *capsule, const char *name)
```

Retrieve the pointer stored in the capsule. On failure, set an exception and return `NULL`.

The name parameter must compare exactly to the name stored in the capsule. If the name stored in the capsule is `NULL`, the name passed in must also be `NULL`. Python uses the C function `strcmp()` to compare capsule names.

```c
PyCapsule_Destructor
PyCapsule_GetDestructor(PyObject *capsule)
```

Return the current destructor stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` destructor. This makes a `NULL` return code somewhat ambiguous; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.

```c
void* PyCapsule_GetContext(PyObject *capsule)
```

Return the current context stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` context. This makes a `NULL` return code somewhat ambiguous; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.
const char* **PyCapsule_GetName**(*PyObject *capsule*)

Return the current name stored in the capsule. On failure, set an exception and return *NULL*.

It is legal for a capsule to have a *NULL* name. This makes a *NULL* return code somewhat ambiguous; use *PyCapsule_IsValid()* or *PyErr_Occurred()* to disambiguate.

**void** *PyCapsule_Import*(const char *name, int no_block)*

Import a pointer to a C object from a capsule attribute in a module. The *name* parameter should specify the full name to the attribute, as in *module.attribute*. The *name* stored in the capsule must match this string exactly. If *no_block* is true, import the module **without** blocking (using *PyImport_ImportModuleNoBlock()*). If *no_block* is false, import the module conventionally (using *PyImport_ImportModule()*).

Return the capsule’s internal **pointer** on success. On failure, set an exception and return *NULL*. However, if *PyCapsule_Import()* failed to import the module, and *no_block* was true, no exception is set.

**int** *PyCapsule_IsValid*(PyObject *capsule, const char *name)*

Determines whether or not *capsule* is a valid capsule. A valid capsule is non-*NULL*, passes *PyCapsule_CheckExact()*), has a non-*NULL* pointer stored in it, and its internal name matches the *name* parameter. (See *PyCapsule_GetPointer()* for information on how capsule names are compared.)

In other words, if *PyCapsule_IsValid()* returns a true value, calls to any of the accessors (any function starting with *PyCapsule_Get()*) are guaranteed to succeed.

Return a nonzero value if the object is valid and matches the
name passed in. Return 0 otherwise. This function will not fail.

int PyCapsule_SetContext(PyObject *capsule, void *context)
Set the context pointer inside capsule to context.

Return 0 on success. Return nonzero and set an exception on failure.

int PyCapsule_SetDestructor(PyObject *capsule, PyCapsule_Destructor destructor)
Set the destructor inside capsule to destructor.

Return 0 on success. Return nonzero and set an exception on failure.

int PyCapsule_SetName(PyObject *capsule, const char *name)
Set the name inside capsule to name. If non-NULL, the name must outlive the capsule. If the previous name stored in the capsule was not NULL, no attempt is made to free it.

Return 0 on success. Return nonzero and set an exception on failure.

int PyCapsule_SetPointer(PyObject *capsule, void *pointer)
Set the void pointer inside capsule to pointer. The pointer may not be NULL.

Return 0 on success. Return nonzero and set an exception on failure.
**CObjects**

**Warning:** The CObject API is deprecated as of Python 2.7. Please switch to the new Capsules API.

**PyCObject**

This subtype of `PyObject` represents an opaque value, useful for C extension modules who need to pass an opaque value (as a `void*` pointer) through Python code to other C code. It is often used to make a C function pointer defined in one module available to other modules, so the regular import mechanism can be used to access C APIs defined in dynamically loaded modules.

```c
int PyCObject_Check(PyObject *p)
    Return true if its argument is a PyCObject.

PyObject* PyCObject_FromVoidPtr(void* cobj, void (*destr)(void *))
    Create a PyCObject from the void * cobj. The destr function will be called when the object is reclaimed, unless it is NULL.

PyObject* PyCObject_FromVoidPtrAndDesc(void* cobj, void* desc, void (*destr)(void *, void *))
    Create a PyCObject from the void * cobj. The destr function will be called when the object is reclaimed. The desc argument can be used to pass extra callback data for the destructor function.

void* PyCObject_AsVoidPtr(PyObject* self)
    Return the object void * that the PyCObject self was created with.

void* PyCObject_GetDesc(PyObject* self)
    Return the description void * that the PyCObject self was created with.
```
with.

int **PyCObject_SetVoidPtr**(PyObject* self, void* cobj)

Set the void pointer inside self to cobj. The PyCObject must not have an associated destructor. Return true on success, false on failure.
Cell Objects

“Cell” objects are used to implement variables referenced by multiple scopes. For each such variable, a cell object is created to store the value; the local variables of each stack frame that references the value contains a reference to the cells from outer scopes which also use that variable. When the value is accessed, the value contained in the cell is used instead of the cell object itself. This de-referencing of the cell object requires support from the generated byte-code; these are not automatically de-referenced when accessed. Cell objects are not likely to be useful elsewhere.

**PyCellObject**

The C structure used for cell objects.

**PyTypeObject** **PyCell_Type**

The type object corresponding to cell objects.

```c
int PyCell_Check(ob)
    Return true if ob is a cell object; ob must not be NULL.
```

```c
PyObject* PyCell_New(PyObject *ob)
    Create and return a new cell object containing the value ob. The parameter may be NULL.
```

```c
PyObject* PyCell_Get(PyObject *cell)
    Return the contents of the cell cell.
```

```c
PyObject* PyCell_GET(PyObject *cell)
    Return value: Borrowed reference.
    Return the contents of the cell cell, but without checking that cell is non-NULL and a cell object.
```
int **cell_set** *(PyObject *cell, PyObject *value)*

Set the contents of the cell object *cell* to *value*. This releases the reference to any current content of the cell. *value* may be *NULL*. *cell* must be non-NULL; if it is not a cell object, `-1` will be returned. On success, `0` will be returned.

void **cell_set** *(PyObject *cell, PyObject *value)*

Sets the value of the cell object *cell* to *value*. No reference counts are adjusted, and no checks are made for safety; *cell* must be non-NULL and must be a cell object.
Generator Objects

Generator objects are what Python uses to implement generator iterators. They are normally created by iterating over a function that yields values, rather than explicitly calling `PyGen_New()`.

**PyGenObject**

The C structure used for generator objects.

**PyTypeObject** PyGen_Type

The type object corresponding to generator objects.

```c
int PyGen_Check(ob)
Return true if ob is a generator object; ob must not be NULL.
```

```c
int PyGen_CheckExact(ob)
Return true if ob's type is PyGen_Type is a generator object; ob must not be NULL.
```

```c
PyObject* PyGen_New(PyFrameObject *frame)
Create and return a new generator object based on the frame object. A reference to frame is stolen by this function. The parameter must not be NULL.
```
Various date and time objects are supplied by the `datetime` module. Before using any of these functions, the header file `datetime.h` must be included in your source (note that this is not included by `Python.h`), and the macro `PyDateTime_IMPORT` must be invoked, usually as part of the module initialisation function. The macro puts a pointer to a C structure into a static variable, `PyDateTimeAPI`, that is used by the following macros.

Type-check macros:

```c
int PyDate_Check(PyObject *ob)
    Return true if `ob` is of type `PyDateTime_DateType` or a subtype of `PyDateTime_DateType`. `ob` must not be `NULL`.

    *New in version 2.4.*

int PyDate_CheckExact(PyObject *ob)
    Return true if `ob` is of type `PyDateTime_DateType`. `ob` must not be `NULL`.

    *New in version 2.4.*

int PyDateTime_Check(PyObject *ob)
    Return true if `ob` is of type `PyDateTime_DateTimeType` or a subtype of `PyDateTime_DateTimeType`. `ob` must not be `NULL`.

    *New in version 2.4.*

int PyDateTime_CheckExact(PyObject *ob)
    Return true if `ob` is of type `PyDateTime_DateTimeType`. `ob` must not be `NULL`.
```
New in version 2.4.

int PyTime_Check(PyObject *ob)
Return true if ob is of type PyDateTime_TimeType or a subtype of PyDateTime_TimeType. ob must not be NULL.

New in version 2.4.

int PyTime_CheckExact(PyObject *ob)
Return true if ob is of type PyDateTime_TimeType. ob must not be NULL.

New in version 2.4.

int PyDelta_Check(PyObject *ob)
Return true if ob is of type PyDateTime_DeltaType or a subtype of PyDateTime_DeltaType. ob must not be NULL.

New in version 2.4.

int PyDelta_CheckExact(PyObject *ob)
Return true if ob is of type PyDateTime_DeltaType. ob must not be NULL.

New in version 2.4.

int PyTZInfo_Check(PyObject *ob)
Return true if ob is of type PyDateTime_TZInfoType or a subtype of PyDateTime_TZInfoType. ob must not be NULL.

New in version 2.4.

int PyTZInfo_CheckExact(PyObject *ob)
Return true if ob is of type PyDateTime_TZInfoType. ob must not be NULL.
New in version 2.4.

Macros to create objects:

**PyObject** `PyDate_FromDate`(int year, int month, int day)

*Return value: New reference.*

Return a `datetime.date` object with the specified year, month and day.

New in version 2.4.

**PyObject** `PyDateTime_FromDateAndTime`(int year, int month, int day, int hour, int minute, int second, int usecond)

*Return value: New reference.*

Return a `datetime.datetime` object with the specified year, month, day, hour, minute, second and microsecond.

New in version 2.4.

**PyObject** `PyTime_FromTime`(int hour, int minute, int second, int usecond)

*Return value: New reference.*

Return a `datetime.time` object with the specified hour, minute, second and microsecond.

New in version 2.4.

**PyObject** `PyDelta_FromDSU`(int days, int seconds, int useconds)

*Return value: New reference.*

Return a `datetime.timedelta` object representing the given number of days, seconds and microseconds. Normalization is performed so that the resulting number of microseconds and seconds lie in the ranges documented for `datetime.timedelta` objects.

New in version 2.4.
Macros to extract fields from date objects. The argument must be an instance of \texttt{PyDateTime\_Date}, including subclasses (such as \texttt{PyDateTime\_DateTime}). The argument must not be \texttt{NULL}, and the type is not checked:

\begin{verbatim}
int PyDateTime\_GET\_YEAR(PyDateTime\_Date *o)
    Return the year, as a positive int.

    \textit{New in version 2.4.}

int PyDateTime\_GET\_MONTH(PyDateTime\_Date *o)
    Return the month, as an int from 1 through 12.

    \textit{New in version 2.4.}

int PyDateTime\_GET\_DAY(PyDateTime\_Date *o)
    Return the day, as an int from 1 through 31.

    \textit{New in version 2.4.}
\end{verbatim}

Macros to extract fields from datetime objects. The argument must be an instance of \texttt{PyDateTime\_DateTime}, including subclasses. The argument must not be \texttt{NULL}, and the type is not checked:

\begin{verbatim}
int PyDateTime\_DATE\_GET\_HOUR(PyDateTime\_DateTime *o)
    Return the hour, as an int from 0 through 23.

    \textit{New in version 2.4.}

int PyDateTime\_DATE\_GET\_MINUTE(PyDateTime\_DateTime *o)
    Return the minute, as an int from 0 through 59.

    \textit{New in version 2.4.}

int PyDateTime\_DATE\_GET\_SECOND(PyDateTime\_DateTime *o)
    Return the second, as an int from 0 through 59.
\end{verbatim}
New in version 2.4.

int PyDateTime_DATE_GET_MICROSECOND(PyDateTime_DateTime *o)
    Return the microsecond, as an int from 0 through 999999.

New in version 2.4.

Macros to extract fields from time objects. The argument must be an instance of PyDateTime_Time, including subclasses. The argument must not be NULL, and the type is not checked:

int PyDateTime_TIME_GET_HOUR(PyDateTime_Time *o)
    Return the hour, as an int from 0 through 23.

New in version 2.4.

int PyDateTime_TIME_GET_MINUTE(PyDateTime_Time *o)
    Return the minute, as an int from 0 through 59.

New in version 2.4.

int PyDateTime_TIME_GET_SECOND(PyDateTime_Time *o)
    Return the second, as an int from 0 through 59.

New in version 2.4.

int PyDateTime_TIME_GET_MICROSECOND(PyDateTime_Time *o)
    Return the microsecond, as an int from 0 through 999999.

New in version 2.4.

Macros for the convenience of modules implementing the DB API:

PyObject* PyDateTime_FromTimestamp(PyObject *args)
    Create and return a new datetime.datetime object given an argument tuple suitable for passing to
datetime.datetime.fromtimestamp().

*New in version 2.4.*

**PyObject** *PyDate_FromTimestamp**(PyObject *args)**

*Return value: New reference.*

Create and return a new `datetime.date` object given an argument tuple suitable for passing to `datetime.date.fromtimestamp()`.

*New in version 2.4.*
Set Objects

New in version 2.5.

This section details the public API for `set` and `frozenset` objects. Any functionality not listed below is best accessed using the either the abstract object protocol (including `PyObject_CallMethod()`, `PyObject_RichCompareBool()`, `PyObject_Hash()`, `PyObject_Repr()`, `PyObject_IsTrue()`, `PyObject_Print()`, and `PyObject_GetIter()`) or the abstract number protocol (including `PyNumber_And()`, `PyNumber_Subtract()`, `PyNumber_Or()`, `PyNumber_Xor()`, `PyNumber_InPlaceAnd()`, `PyNumber_InPlaceSubtract()`, `PyNumber_InPlaceOr()`, and `PyNumber_InPlaceXor()`).

**PySetObject**

This subtype of `PyObject` is used to hold the internal data for both `set` and `frozenset` objects. It is like a `PyDictObject` in that it is a fixed size for small sets (much like tuple storage) and will point to a separate, variable sized block of memory for medium and large sized sets (much like list storage). None of the fields of this structure should be considered public and are subject to change. All access should be done through the documented API rather than by manipulating the values in the structure.

**PyTypeObject** **PySet_Type**

This is an instance of `PyTypeObject` representing the Python `set` type.

**PyTypeObject** **PyFrozenSet_Type**

This is an instance of `PyTypeObject` representing the Python `frozenset` type.

The following type check macros work on pointers to any Python
object. Likewise, the constructor functions work with any iterable Python object.

```c
int PySet_Check(PyObject *p)
```

Return true if `p` is a `set` object or an instance of a subtype.

*New in version 2.6.*

```c
int PyFrozenSet_Check(PyObject *p)
```

Return true if `p` is a `frozenset` object or an instance of a subtype.

*New in version 2.6.*

```c
int PyAnySet_Check(PyObject *p)
```

Return true if `p` is a `set` object, a `frozenset` object, or an instance of a subtype.

```c
int PyAnySet_CheckExact(PyObject *p)
```

Return true if `p` is a `set` object or a `frozenset` object but not an instance of a subtype.

```c
int PyFrozenSet_CheckExact(PyObject *p)
```

Return true if `p` is a `frozenset` object but not an instance of a subtype.

```c
PyObject* PySet_New(PyObject *iterable)
```

*Return value: New reference.*

Return a new `set` containing objects returned by the `iterable`. The `iterable` may be `NULL` to create a new empty set. Return the new set on success or `NULL` on failure. Raise `TypeError` if `iterable` is not actually iterable. The constructor is also useful for copying a set (`c=set(s)`).

```c
PyObject* PyFrozenSet_New(PyObject *iterable)
```

*Return value: New reference.*
Return a new `frozenset` containing objects returned by the `iterable`. The `iterable` may be `NULL` to create a new empty frozenset. Return the new set on success or `NULL` on failure. Raise `TypeError` if `iterable` is not actually iterable.

* Changed in version 2.6: * Now guaranteed to return a brand-new `frozenset`. Formerly, frozensets of zero-length were a singleton. This got in the way of building-up new frozensets with `PySet_Add()`.

The following functions and macros are available for instances of `set` or `frozenset` or instances of their subtypes.

- **Py_ssize_t PySet_Size(PyObject *anyset)**
  Return the length of a `set` or `frozenset` object. Equivalent to `len(anyset)`. Raises a `PyExc_SystemError` if `anyset` is not a `set`, `frozenset`, or an instance of a subtype.

  * Changed in version 2.5: * This function returned an `int`. This might require changes in your code for properly supporting 64-bit systems.

- **Py_ssize_t PySet_GET_SIZE(PyObject *anyset)**
  Macro form of `PySet_Size()` without error checking.

- **int PySet_Contains(PyObject *anyset, PyObject *key)**
  Return 1 if found, 0 if not found, and -1 if an error is encountered. Unlike the Python `__contains__()` method, this function does not automatically convert unhashable sets into temporary frozensets. Raise a `TypeError` if the `key` is unhashable. Raise `PyExc_SystemError` if `anyset` is not a `set`, `frozenset`, or an instance of a subtype.

- **int PySet_Add(PyObject *set, PyObject *key)**
Add key to a set instance. Does not apply to frozenset instances. Return 0 on success or -1 on failure. Raise a TypeError if the key is unhashable. Raise a MemoryError if there is no room to grow. Raise a SystemError if set is not an instance of set or its subtype.

*Changed in version 2.6:* Now works with instances of frozenset or its subtypes. Like PyTuple_SetItem() in that it can be used to fill-in the values of brand new frozensets before they are exposed to other code.

The following functions are available for instances of set or its subtypes but not for instances of frozenset or its subtypes.

```c
int PySet_Discard(PyObject *set, PyObject *key)
```

Return 1 if found and removed, 0 if not found (no action taken), and -1 if an error is encountered. Does not raise KeyError for missing keys. Raise a TypeError if the key is unhashable. Unlike the Python discard() method, this function does not automatically convert unhashable sets into temporary frozensets. Raise PyExc_SystemError if set is not an instance of set or its subtype.

```c
PyObject* PySet_Pop(PyObject *set)
```


Return a new reference to an arbitrary object in the set, and removes the object from the set. Return NULL on failure. Raise KeyError if the set is empty. Raise a SystemError if set is not an instance of set or its subtype.

```c
int PySet_Clear(PyObject *set)
```

Empty an existing set of all elements.
Code Objects

Code objects are a low-level detail of the CPython implementation. Each one represents a chunk of executable code that hasn't yet been bound into a function.

**PyCodeObject**

The C structure of the objects used to describe code objects. The fields of this type are subject to change at any time.

**PyTypeObject PyCode_Type**

This is an instance of `PyTypeObject` representing the Python code type.

```c
int PyCode_Check(PyObject *co)
Return true if co is a code object.
```

```c
int PyCode_GetNumFree(PyObject *co)
Return the number of free variables in co.
```

```c
PyCodeObject *PyCode_New(int argcount, int nlocals, int stacksize, int flags, PyObject *code, PyObject *consts, PyObject *names, PyObject *varnames, PyObject *freevars, PyObject *cellvars, PyObject *filename, PyObject *name, int firstlineno, PyCodeObject *lnotab)
Return a new code object. If you need a dummy code object to create a frame, use PyCode_NewEmpty() instead. Calling PyCode_New() directly can bind you to a precise Python version since the definition of the bytecode changes often.
```

```c
int PyCode_NewEmpty(const char *filename, const char *funcname, int firstlineno)
Return a new empty code object with the specified filename, function name, and first line number. It is illegal to exec or eval() the resulting code object.
```
Initialization, Finalization, and Threads
Initializing and finalizing the interpreter

void **Py_Initialize**

Initialize the Python interpreter. In an application embedding Python, this should be called before using any other Python/C API functions; with the exception of **Py_SetProgramName()**, **Py_SetPythonHome()**, **PyEval_InitThreads()**, **PyEval_ReleaseLock()**, and **PyEval_AcquireLock()**. This initializes the table of loaded modules (**sys.modules**), and creates the fundamental modules **__builtin__**, **__main__** and **sys**. It also initializes the module search path (**sys.path**). It does not set **sys.argv**; use **PySys_SetArgvEx()** for that. This is a no-op when called for a second time (without calling **Py_Finalize()** first). There is no return value; it is a fatal error if the initialization fails.

void **Py_InitializeEx**(int **initsigs**)

This function works like **Py_Initialize()** if **initsigs** is 1. If **initsigs** is 0, it skips initialization registration of signal handlers, which might be useful when Python is embedded.

*New in version 2.4.*

int **Py_IsInitialized**

Return true (nonzero) when the Python interpreter has been initialized, false (zero) if not. After **Py_Finalize()** is called, this returns false until **Py_Initialize()** is called again.

void **Py_Finalize**

Undo all initializations made by **Py_Initialize()** and subsequent use of Python/C API functions, and destroy all sub-interpreters (see **Py_NewInterpreter()** below) that were created and not yet destroyed since the last call to **Py_Initialize()**. Ideally, this frees
all memory allocated by the Python interpreter. This is a no-op when called for a second time (without calling `Py_Initialize()` again first). There is no return value; errors during finalization are ignored.

This function is provided for a number of reasons. An embedding application might want to restart Python without having to restart the application itself. An application that has loaded the Python interpreter from a dynamically loadable library (or DLL) might want to free all memory allocated by Python before unloading the DLL. During a hunt for memory leaks in an application a developer might want to free all memory allocated by Python before exiting from the application.

**Bugs and caveats:** The destruction of modules and objects in modules is done in random order; this may cause destructors (\_\_\_del\_\_\_() methods) to fail when they depend on other objects (even functions) or modules. Dynamically loaded extension modules loaded by Python are not unloaded. Small amounts of memory allocated by the Python interpreter may not be freed (if you find a leak, please report it). Memory tied up in circular references between objects is not freed. Some memory allocated by extension modules may not be freed. Some extensions may not work properly if their initialization routine is called more than once; this can happen if an application calls `Py_Initialize()` and `Py_Finalize()` more than once.
Process-wide parameters

void Py_SetProgramName(char *name)

This function should be called before Py_Initialize() is called for the first time, if it is called at all. It tells the interpreter the value of the argv[0] argument to the main() function of the program. This is used by Py_GetPath() and some other functions below to find the Python run-time libraries relative to the interpreter executable. The default value is 'python'. The argument should point to a zero-terminated character string in static storage whose contents will not change for the duration of the program's execution. No code in the Python interpreter will change the contents of this storage.

char* Py_GetProgramName()

Return the program name set with Py_SetProgramName(), or the default. The returned string points into static storage; the caller should not modify its value.

char* Py_GetPrefix()

Return the prefix for installed platform-independent files. This is derived through a number of complicated rules from the program name set with Py_SetProgramName() and some environment variables; for example, if the program name is '/usr/local/bin/python', the prefix is '/usr/local'. The returned string points into static storage; the caller should not modify its value. This corresponds to the prefix variable in the top-level Makefile and the --prefix argument to the configure script at build time. The value is available to Python code as sys.prefix. It is only useful on Unix. See also the next function.

char* Py_GetExecPrefix()
Return the `exec-prefix` for installed platform-dependent files. This is derived through a number of complicated rules from the program name set with `Py_SetProgramName()` and some environment variables; for example, if the program name is ‘/usr/local/bin/python’, the exec-prefix is ‘/usr/local’. The returned string points into static storage; the caller should not modify its value. This corresponds to the `exec_prefix` variable in the top-level Makefile and the `--exec-prefix` argument to the `configure` script at build time. The value is available to Python code as `sys.exec_prefix`. It is only useful on Unix.

Background: The exec-prefix differs from the prefix when platform dependent files (such as executables and shared libraries) are installed in a different directory tree. In a typical installation, platform dependent files may be installed in the `/usr/local/plat` subtree while platform independent may be installed in `/usr/local`.

Generally speaking, a platform is a combination of hardware and software families, e.g. Sparc machines running the Solaris 2.x operating system are considered the same platform, but Intel machines running Solaris 2.x are another platform, and Intel machines running Linux are yet another platform. Different major revisions of the same operating system generally also form different platforms. Non-Unix operating systems are a different story; the installation strategies on those systems are so different that the prefix and exec-prefix are meaningless, and set to the empty string. Note that compiled Python bytecode files are platform independent (but not independent from the Python version by which they were compiled!).

System administrators will know how to configure the `mount` or `automount` programs to share `/usr/local` between platforms while having `/usr/local/plat` be a different filesystem for each
platform.

char* Py_GetProgramFullPath()
Return the full program name of the Python executable; this is computed as a side-effect of deriving the default module search path from the program name (set by Py_SetProgramName() above). The returned string points into static storage; the caller should not modify its value. The value is available to Python code as sys.executable.

char* Py_GetPath()
Return the default module search path; this is computed from the program name (set by Py_SetProgramName() above) and some environment variables. The returned string consists of a series of directory names separated by a platform dependent delimiter character. The delimiter character is ':' on Unix and Mac OS X, ';' on Windows. The returned string points into static storage; the caller should not modify its value. The list sys.path is initialized with this value on interpreter startup; it can be (and usually is) modified later to change the search path for loading modules.

const char* Py_GetVersion()
Return the version of this Python interpreter. This is a string that looks something like

"1.5 (#67, Dec 31 1997, 22:34:28) [GCC 2.7.2.2]"

The first word (up to the first space character) is the current Python version; the first three characters are the major and minor version separated by a period. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as sys.version.

const char* Py_GetPlatform()
Return the platform identifier for the current platform. On Unix, this is formed from the “official” name of the operating system, converted to lower case, followed by the major revision number; e.g., for Solaris 2.x, which is also known as SunOS 5.x, the value is 'sunos5'. On Mac OS X, it is 'darwin'. On Windows, it is 'win'. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.platform`.

`const char* Py_GetCopyright()`
Return the official copyright string for the current Python version, for example

'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.copyright`.

`const char* Py_GetCompiler()`
Return an indication of the compiler used to build the current Python version, in square brackets, for example:

"[GCC 2.7.2.2]"

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

`const char* Py_GetBuildInfo()`
Return information about the sequence number and build date and time of the current Python interpreter instance, for example

"#67, Aug 1 1997, 22:34:28"
The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

```c
void PySys_SetArgvEx(int argc, char **argv, int updatepath)
```

Set `sys.argv` based on `argc` and `argv`. These parameters are similar to those passed to the program’s `main()` function with the difference that the first entry should refer to the script file to be executed rather than the executable hosting the Python interpreter. If there isn’t a script that will be run, the first entry in `argv` can be an empty string. If this function fails to initialize `sys.argv`, a fatal condition is signalled using `Py_FatalError()`.

If `updatepath` is zero, this is all the function does. If `updatepath` is non-zero, the function also modifies `sys.path` according to the following algorithm:

- If the name of an existing script is passed in `argv[0]`, the absolute path of the directory where the script is located is prepended to `sys.path`.
- Otherwise (that is, if `argc` is 0 or `argv[0]` doesn’t point to an existing file name), an empty string is prepended to `sys.path`, which is the same as prepending the current working directory `"."`.

**Note:** It is recommended that applications embedding the Python interpreter for purposes other than executing a single script pass 0 as `updatepath`, and update `sys.path` themselves if desired. See CVE-2008-5983.

On versions before 2.6.6, you can achieve the same effect by manually popping the first `sys.path` element after having called `PySys_SetArgv()`, for example using:

```python
PyRun_SimpleString("import sys; sys.path.pop(0)\n");
```
New in version 2.6.6.

void PySys_SetArgv(int argc, char **argv)
   This function works like PySys_SetArgvEx() with updatepath set to 1.

void Py_SetPythonHome(char *home)
   Set the default “home” directory, that is, the location of the standard Python libraries. See PYTHONHOME for the meaning of the argument string.

   The argument should point to a zero-terminated character string in static storage whose contents will not change for the duration of the program’s execution. No code in the Python interpreter will change the contents of this storage.

char* Py_GetPythonHome()
   Return the default “home”, that is, the value set by a previous call to Py_SetPythonHome(), or the value of the PYTHONHOME environment variable if it is set.
Thread State and the Global Interpreter Lock

The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the global interpreter lock or GIL, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

Therefore, the rule exists that only the thread that has acquired the GIL may operate on Python objects or call Python/C API functions. In order to emulate concurrency of execution, the interpreter regularly tries to switch threads (see `sys.setcheckinterval()`). The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.

The Python interpreter keeps some thread-specific bookkeeping information inside a data structure called `PyThreadState`. There's also one global variable pointing to the current `PyThreadState`: it can be retrieved using `PyThreadState_Get()`.

Releasing the GIL from extension code

Most extension code manipulating the GIL has the following simple structure:

Save the `thread` state in a local variable.
Release the global interpreter lock.
... Do some blocking I/O operation ...
Reacquire the global interpreter lock.
Restore the thread state from the local variable.

This is so common that a pair of macros exists to simplify it:

\[
\begin{array}{l}
\text{Py\_BEGIN\_ALLOW\_THREADS} \\
\ldots \text{Do some blocking I/O operation} \ldots \\
\text{Py\_END\_ALLOW\_THREADS}
\end{array}
\]

The \texttt{Py\_BEGIN\_ALLOW\_THREADS} macro opens a new block and declares a hidden local variable; the \texttt{Py\_END\_ALLOW\_THREADS} macro closes the block. These two macros are still available when Python is compiled without thread support (they simply have an empty expansion).

When thread support is enabled, the block above expands to the following code:

\[
\begin{array}{l}
\text{PyThreadState \_*\_save;} \\
\_save = \text{PyEval\_SaveThread();} \\
\ldots \text{Do some blocking I/O operation...} \\
\text{PyEval\_RestoreThread(_save);} \\
\end{array}
\]

Here is how these functions work: the global interpreter lock is used to protect the pointer to the current thread state. When releasing the lock and saving the thread state, the current thread state pointer must be retrieved before the lock is released (since another thread could immediately acquire the lock and store its own thread state in the global variable). Conversely, when acquiring the lock and restoring the thread state, the lock must be acquired before storing the thread state pointer.

\textbf{Note: } Calling system I/O functions is the most common use case for releasing the GIL, but it can also be useful before calling long-running computations which don’t need access to Python objects, such as compression or cryptographic functions operating over memory buffers. For example, the standard \texttt{zlib} and \texttt{hashlib}.
modules release the GIL when compressing or hashing data.

## Non-Python created threads

When threads are created using the dedicated Python APIs (such as the `threading` module), a thread state is automatically associated to them and the code showed above is therefore correct. However, when threads are created from C (for example by a third-party library with its own thread management), they don't hold the GIL, nor is there a thread state structure for them.

If you need to call Python code from these threads (often this will be part of a callback API provided by the aforementioned third-party library), you must first register these threads with the interpreter by creating a thread state data structure, then acquiring the GIL, and finally storing their thread state pointer, before you can start using the Python/C API. When you are done, you should reset the thread state pointer, release the GIL, and finally free the thread state data structure.

The `PyGILState_Ensure()` and `PyGILState_Release()` functions do all of the above automatically. The typical idiom for calling into Python from a C thread is:

```c
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();

/* Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */

/* Release the thread. No Python API allowed beyond this point. */
PyGILState_Release(gstate);
```

Note that the `PyGILState_*()` functions assume there is only one global interpreter (created automatically by `Py_Initialize()`). Python
supports the creation of additional interpreters (using `Py_NewInterpreter()`), but mixing multiple interpreters and the `PyGILState_*()` API is unsupported.

Another important thing to note about threads is their behaviour in the face of the C `fork()` call. On most systems with `fork()`, after a process forks only the thread that issued the fork will exist. That also means any locks held by other threads will never be released. Python solves this for `os.fork()` by acquiring the locks it uses internally before the fork, and releasing them afterwards. In addition, it resets any `Lock Objects` in the child. When extending or embedding Python, there is no way to inform Python of additional (non-Python) locks that need to be acquired before or reset after a fork. OS facilities such as `pthread_atfork()` would need to be used to accomplish the same thing. Additionally, when extending or embedding Python, calling `fork()` directly rather than through `os.fork()` (and returning to or calling into Python) may result in a deadlock by one of Python’s internal locks being held by a thread that is defunct after the fork. `PyOS_AfterFork()` tries to reset the necessary locks, but is not always able to.

### High-level API

These are the most commonly used types and functions when writing C extension code, or when embedding the Python interpreter:

**PyInterpreterState**

This data structure represents the state shared by a number of cooperating threads. Threads belonging to the same interpreter share their module administration and a few other internal items. There are no public members in this structure.

Threads belonging to different interpreters initially share nothing, except process state like available memory, open file descriptors
and such. The global interpreter lock is also shared by all threads, regardless of to which interpreter they belong.

**PyThreadState**

This data structure represents the state of a single thread. The only public data member is `PyInterpreterState *interp`, which points to this thread's interpreter state.

**void PyEval_InitThreads()**

Initialize and acquire the global interpreter lock. It should be called in the main thread before creating a second thread or engaging in any other thread operations such as `PyEval_ReleaseLock()` or `PyEval_ReleaseThread(tstate)`. It is not needed before calling `PyEval_SaveThread()` or `PyEval_RestoreThread()`.

This is a no-op when called for a second time. It is safe to call this function before calling `Py_Initialize()`.

**Note:** When only the main thread exists, no GIL operations are needed. This is a common situation (most Python programs do not use threads), and the lock operations slow the interpreter down a bit. Therefore, the lock is not created initially. This situation is equivalent to having acquired the lock: when there is only a single thread, all object accesses are safe. Therefore, when this function initializes the global interpreter lock, it also acquires it. Before the Python _thread module creates a new thread, knowing that either it has the lock or the lock hasn’t been created yet, it calls `PyEval_InitThreads()`. When this call returns, it is guaranteed that the lock has been created and that the calling thread has acquired it. It is **not** safe to call this function when it is unknown which thread (if any) currently has the global interpreter lock.

This function is not available when thread support is disabled at
int PyEval_ThreadsInitialized()

Returns a non-zero value if PyEval_InitThreads() has been called. This function can be called without holding the GIL, and therefore can be used to avoid calls to the locking API when running single-threaded. This function is not available when thread support is disabled at compile time.

New in version 2.4.

PyThreadState* PyEval_SaveThread()

Release the global interpreter lock (if it has been created and thread support is enabled) and reset the thread state to NULL, returning the previous thread state (which is not NULL). If the lock has been created, the current thread must have acquired it. (This function is available even when thread support is disabled at compile time.)

void PyEval_RestoreThread(PyThreadState *tstate)

Acquire the global interpreter lock (if it has been created and thread support is enabled) and set the thread state to tstate, which must not be NULL. If the lock has been created, the current thread must not have acquired it, otherwise deadlock ensues. (This function is available even when thread support is disabled at compile time.)

PyThreadState* PyThreadState_Get()

Return the current thread state. The global interpreter lock must be held. When the current thread state is NULL, this issues a fatal error (so that the caller needn’t check for NULL).

PyThreadState* PyThreadState_Swap(PyThreadState *tstate)

Swap the current thread state with the thread state given by the argument tstate, which may be NULL. The global interpreter lock
must be held and is not released.

void PyEval_ReInitThreads()
    This function is called from PyOS_AfterFork() to ensure that newly created child processes don't hold locks referring to threads which are not running in the child process.

The following functions use thread-local storage, and are not compatible with sub-interpreters:

PyGILState_STATE PyGILState_Ensure()
    Ensure that the current thread is ready to call the Python C API regardless of the current state of Python, or of the global interpreter lock. This may be called as many times as desired by a thread as long as each call is matched with a call to PyGILState_Release(). In general, other thread-related APIs may be used between PyGILState_Ensure() and PyGILState_Release() calls as long as the thread state is restored to its previous state before the Release(). For example, normal usage of the Py_BEGIN_ALLOW_THREADS and Py_END_ALLOW_THREADS macros is acceptable.

    The return value is an opaque “handle” to the thread state when PyGILState_Ensure() was called, and must be passed to PyGILState_Release() to ensure Python is left in the same state. Even though recursive calls are allowed, these handles cannot be shared - each unique call to PyGILState_Ensure() must save the handle for its call to PyGILState_Release().

    When the function returns, the current thread will hold the GIL and be able to call arbitrary Python code. Failure is a fatal error.

    New in version 2.3.

void PyGILState_Release(PyGILState_STATE)
Release any resources previously acquired. After this call, Python’s state will be the same as it was prior to the corresponding `PyGILState_Ensure()` call (but generally this state will be unknown to the caller, hence the use of the GILState API).

Every call to `PyGILState_Ensure()` must be matched by a call to `PyGILState_Release()` on the same thread.

*New in version 2.3.*

`PyThreadState* PyGILState_GetThisThreadState()`

Get the current thread state for this thread. May return `NULL` if no GILState API has been used on the current thread. Note that the main thread always has such a thread-state, even if no auto-thread-state call has been made on the main thread. This is mainly a helper/diagnostic function.

*New in version 2.3.*

The following macros are normally used without a trailing semicolon; look for example usage in the Python source distribution.

`Py_BEGIN_ALLOW_THREADS`

This macro expands to `{ PyThreadState *_save; _save = PyEval_SaveThread(); }`. Note that it contains an opening brace; it must be matched with a following `Py_END_ALLOW_THREADS` macro. See above for further discussion of this macro. It is a no-op when thread support is disabled at compile time.

`Py_END_ALLOW_THREADS`

This macro expands to `PyEval_RestoreThread(_save); }`. Note that it contains a closing brace; it must be matched with an earlier `Py_BEGIN_ALLOW_THREADS` macro. See above for further discussion of this macro. It is a no-op when thread support is disabled at compile time.
**Py_BLOCK_THREADS**

This macro expands to `PyEval_RestoreThread(_save);`: it is equivalent to `Py_END_ALLOW_THREADS` without the closing brace. It is a no-op when thread support is disabled at compile time.

**Py_UNBLOCK_THREADS**

This macro expands to `_save = PyEval_SaveThread();`: it is equivalent to `Py_BEGIN_ALLOW_THREADS` without the opening brace and variable declaration. It is a no-op when thread support is disabled at compile time.

### Low-level API

All of the following functions are only available when thread support is enabled at compile time, and must be called only when the global interpreter lock has been created.

**PyInterpreterState* PyInterpreterState_New()**

Create a new interpreter state object. The global interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

**void PyInterpreterState_Clear(PyInterpreterState *interp)**

Reset all information in an interpreter state object. The global interpreter lock must be held.

**void PyInterpreterState_Delete(PyInterpreterState *interp)**

Destroy an interpreter state object. The global interpreter lock need not be held. The interpreter state must have been reset with a previous call to `PyInterpreterState_Clear()`.

**PyThreadState* PyThreadState_New(PyInterpreterState *interp)**

Create a new thread state object belonging to the given interpreter object. The global interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.
void PyThreadState_Clear(PyThreadState *tstate)
    Reset all information in a thread state object. The global interpreter lock must be held.

void PyThreadState_Delete(PyThreadState *tstate)
    Destroy a thread state object. The global interpreter lock need not be held. The thread state must have been reset with a previous call to PyThreadState_Clear().

PyObject* PyThreadState_GetDict()
    Return value: Borrowed reference.
    Return a dictionary in which extensions can store thread-specific state information. Each extension should use a unique key to use to store state in the dictionary. It is okay to call this function when no current thread state is available. If this function returns NULL, no exception has been raised and the caller should assume no current thread state is available.

    Changed in version 2.3: Previously this could only be called when a current thread is active, and NULL meant that an exception was raised.

int PyThreadState_SetAsyncExc(long id, PyObject *exc)
    Asynchronously raise an exception in a thread. The id argument is the thread id of the target thread; exc is the exception object to be raised. This function does not steal any references to exc. To prevent naive misuse, you must write your own C extension to call this. Must be called with the GIL held. Returns the number of thread states modified; this is normally one, but will be zero if the thread id isn’t found. If exc is NULL, the pending exception (if any) for the thread is cleared. This raises no exceptions.

    New in version 2.3.

void PyEval_AcquireThread(PyThreadState *tstate)
Acquire the global interpreter lock and set the current thread state to \textit{tstate}, which should not be \textit{NULL}. The lock must have been created earlier. If this thread already has the lock, deadlock ensues.

\textbf{PyEval_RestoreThread()} is a higher-level function which is always available (even when thread support isn’t enabled or when threads have not been initialized).

\textbf{void PyEval_ReleaseThread(PyThreadState \textit{tstate})}
Reset the current thread state to \textit{NULL} and release the global interpreter lock. The lock must have been created earlier and must be held by the current thread. The \textit{tstate} argument, which must not be \textit{NULL}, is only used to check that it represents the current thread state — if it isn’t, a fatal error is reported.

\textbf{PyEval_SaveThread()} is a higher-level function which is always available (even when thread support isn’t enabled or when threads have not been initialized).

\textbf{void PyEval_AcquireLock()}

Acquire the global interpreter lock. The lock must have been created earlier. If this thread already has the lock, a deadlock ensues.

\textbf{Warning:} This function does not change the current thread state. Please use \textbf{PyEval_RestoreThread()} or \textbf{PyEval_AcquireThread()} instead.

\textbf{void PyEval_ReleaseLock()}

Release the global interpreter lock. The lock must have been created earlier.

\textbf{Warning:} This function does not change the current thread
Please use `PyEval_SaveThread()` or `PyEval_ReleaseThread()` instead.
Sub-interpreter support

While in most uses, you will only embed a single Python interpreter, there are cases where you need to create several independent interpreters in the same process and perhaps even in the same thread. Sub-interpreters allow you to do that. You can switch between sub-interpreters using the `PyThreadState_Swap()` function. You can create and destroy them using the following functions:

**PyThreadState**\* `Py_NewInterpreter()`  
Create a new sub-interpreter. This is an (almost) totally separate environment for the execution of Python code. In particular, the new interpreter has separate, independent versions of all imported modules, including the fundamental modules `builtins`, `__main__` and `sys`. The table of loaded modules (`sys.modules`) and the module search path (`sys.path`) are also separate. The new environment has no `sys.argv` variable. It has new standard I/O stream file objects `sys.stdin`, `sys.stdout` and `sys.stderr` (however these refer to the same underlying file descriptors).

The return value points to the first thread state created in the new sub-interpreter. This thread state is made in the current thread state. Note that no actual thread is created; see the discussion of thread states below. If creation of the new interpreter is unsuccessful, `NULL` is returned; no exception is set since the exception state is stored in the current thread state and there may not be a current thread state. (Like all other Python/C API functions, the global interpreter lock must be held before calling this function and is still held when it returns; however, unlike most other Python/C API functions, there needn’t be a current thread state on entry.)

Extension modules are shared between (sub-)interpreters as
follows: the first time a particular extension is imported, it is initialized normally, and a (shallow) copy of its module’s dictionary is squirreled away. When the same extension is imported by another (sub-)interpreter, a new module is initialized and filled with the contents of this copy; the extension’s `init` function is not called. Note that this is different from what happens when an extension is imported after the interpreter has been completely re-initialized by calling `Py_Finalize()` and `Py_Initialize()`; in that case, the extension's `initmodule` function is called again.

```c
void Py_EndInterpreter(PyThreadState *tstate)
```

Destroy the (sub-)interpreter represented by the given thread state. The given thread state must be the current thread state. See the discussion of thread states below. When the call returns, the current thread state is `NULL`. All thread states associated with this interpreter are destroyed. (The global interpreter lock must be held before calling this function and is still held when it returns.) `Py_Finalize()` will destroy all sub-interpreters that haven’t been explicitly destroyed at that point.

## Bugs and caveats

Because sub-interpreters (and the main interpreter) are part of the same process, the insulation between them isn’t perfect — for example, using low-level file operations like `os.close()` they can (accidentally or maliciously) affect each other’s open files. Because of the way extensions are shared between (sub-)interpreters, some extensions may not work properly; this is especially likely when the extension makes use of (static) global variables, or when the extension manipulates its module’s dictionary after its initialization. It is possible to insert objects created in one sub-interpreter into a namespace of another sub-interpreter; this should be done with great care to avoid sharing user-defined functions, methods,
instances or classes between sub-interpreters, since import operations executed by such objects may affect the wrong (sub-)interpreter’s dictionary of loaded modules.

Also note that combining this functionality with `PyGILState_*()` APIs is delicate, because these APIs assume a bijection between Python thread states and OS-level threads, an assumption broken by the presence of sub-interpreters. It is highly recommended that you don’t switch sub-interpreters between a pair of matching `PyGILState_Engure()` and `PyGILState_Release()` calls. Furthermore, extensions (such as `ctypes`) using these APIs to allow calling of Python code from non-Python created threads will probably be broken when using sub-interpreters.
Asynchronous Notifications

A mechanism is provided to make asynchronous notifications to the main interpreter thread. These notifications take the form of a function pointer and a void pointer argument.

```c
int Py_AddPendingCall(int (*func)(void *), void *arg)
```

Schedule a function to be called from the main interpreter thread. On success, 0 is returned and `func` is queued for being called in the main thread. On failure, -1 is returned without setting any exception.

When successfully queued, `func` will be eventually called from the main interpreter thread with the argument `arg`. It will be called asynchronously with respect to normally running Python code, but with both these conditions met:

- on a bytecode boundary;
- with the main thread holding the global interpreter lock (`func` can therefore use the full C API).

`func` must return 0 on success, or -1 on failure with an exception set. `func` won’t be interrupted to perform another asynchronous notification recursively, but it can still be interrupted to switch threads if the global interpreter lock is released.

This function doesn’t need a current thread state to run, and it doesn’t need the global interpreter lock.

**Warning:** This is a low-level function, only useful for very special cases. There is no guarantee that `func` will be called as quick as possible. If the main thread is busy executing a system call, `func` won’t be called before the system call returns. This function is generally not suitable for calling Python code from
arbitrary C threads. Instead, use the PyGILState API.

New in version 2.7.
Profiling and Tracing

The Python interpreter provides some low-level support for attaching profiling and execution tracing facilities. These are used for profiling, debugging, and coverage analysis tools.

Starting with Python 2.2, the implementation of this facility was substantially revised, and an interface from C was added. This C interface allows the profiling or tracing code to avoid the overhead of calling through Python-level callable objects, making a direct C function call instead. The essential attributes of the facility have not changed; the interface allows trace functions to be installed per-thread, and the basic events reported to the trace function are the same as had been reported to the Python-level trace functions in previous versions.

```c
int (*Py_tracefunc)(PyObject *obj, PyFrameObject *frame, int what, PyObject *arg)
```

The type of the trace function registered using `PyEval_SetProfile()` and `PyEval_SetTrace()`. The first parameter is the object passed to the registration function as `obj`, `frame` is the frame object to which the event pertains, `what` is one of the constants `PyTrace_CALL`, `PyTrace_EXCEPTION`, `PyTrace_LINE`, `PyTrace_RETURN`, `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION`, or `PyTrace_C_RETURN`, and `arg` depends on the value of `what`:

<table>
<thead>
<tr>
<th>Value of what</th>
<th>Meaning of arg</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PyTrace_CALL</code></td>
<td>Always <code>NULL</code>.</td>
</tr>
<tr>
<td><code>PyTrace_EXCEPTION</code></td>
<td>Exception information as returned by <code>sys.exc_info()</code></td>
</tr>
<tr>
<td><code>PyTrace_LINE</code></td>
<td>Always <code>NULL</code>.</td>
</tr>
<tr>
<td><code>PyTrace_RETURN</code></td>
<td>Value being returned to the caller, or <code>NULL</code> if caused by an exception.</td>
</tr>
<tr>
<td>PyTrace_C_CALL</td>
<td>Function object being called.</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>PyTrace_C_EXCEPTION</td>
<td>Function object being called.</td>
</tr>
<tr>
<td>PyTrace_C_RETURN</td>
<td>Function object being called.</td>
</tr>
</tbody>
</table>

**int PyTrace_CALL**
The value of the *what* parameter to a `Py_tracefunc` function when a new call to a function or method is being reported, or a new entry into a generator. Note that the creation of the iterator for a generator function is not reported as there is no control transfer to the Python bytecode in the corresponding frame.

**int PyTrace_EXCEPTION**
The value of the *what* parameter to a `Py_tracefunc` function when an exception has been raised. The callback function is called with this value for *what* when after any bytecode is processed after which the exception becomes set within the frame being executed. The effect of this is that as exception propagation causes the Python stack to unwind, the callback is called upon return to each frame as the exception propagates. Only trace functions receives these events; they are not needed by the profiler.

**int PyTrace_LINE**
The value passed as the *what* parameter to a trace function (but not a profiling function) when a line-number event is being reported.

**int PyTrace_RETURN**
The value for the *what* parameter to `Py_tracefunc` functions when a call is returning without propagating an exception.

**int PyTrace_C_CALL**
The value for the *what* parameter to `Py_tracefunc` functions when a C function is about to be called.
int PyTrace_C_EXCEPTION
The value for the what parameter to Py_tracefunc functions when a C function has raised an exception.

int PyTrace_C_RETURN
The value for the what parameter to Py_tracefunc functions when a C function has returned.

void PyEval_SetProfile(Py_tracefunc func, PyObject *obj)
Set the profiler function to func. The obj parameter is passed to the function as its first parameter, and may be any Python object, or NULL. If the profile function needs to maintain state, using a different value for obj for each thread provides a convenient and thread-safe place to store it. The profile function is called for all monitored events except the line-number events.

void PyEval_SetTrace(Py_tracefunc func, PyObject *obj)
Set the tracing function to func. This is similar to PyEval_SetProfile(), except the tracing function does receive line-number events.

PyObject* PyEval_GetCallStats(PyObject *self)
Return a tuple of function call counts. There are constants defined for the positions within the tuple:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCALL_ALL</td>
<td>0</td>
</tr>
<tr>
<td>PCALL_FUNCTION</td>
<td>1</td>
</tr>
<tr>
<td>PCALL_FAST_FUNCTION</td>
<td>2</td>
</tr>
<tr>
<td>PCALL_FASTER_FUNCTION</td>
<td>3</td>
</tr>
<tr>
<td>PCALL_METHOD</td>
<td>4</td>
</tr>
<tr>
<td>PCALL_BOUND_METHOD</td>
<td>5</td>
</tr>
<tr>
<td>PCALL_CFUNCTION</td>
<td>6</td>
</tr>
<tr>
<td>PCALL_TYPE</td>
<td>7</td>
</tr>
<tr>
<td>PCALL_GENERATOR</td>
<td>8</td>
</tr>
<tr>
<td>-----------------</td>
<td>---</td>
</tr>
<tr>
<td>PCALL_OTHER</td>
<td>9</td>
</tr>
<tr>
<td>PCALL_POP</td>
<td>10</td>
</tr>
</tbody>
</table>

**PCALL_FAST_FUNCTION** means no argument tuple needs to be created. **PCALL_FASTER_FUNCTION** means that the fast-path frame setup code is used.

If there is a method call where the call can be optimized by changing the argument tuple and calling the function directly, it gets recorded twice.

This function is only present if Python is compiled with **CALL_PROFILE** defined.
Advanced Debugger Support

These functions are only intended to be used by advanced debugging tools.

**PyInterpreterState**

*PyInterpreterState* Head()

Return the interpreter state object at the head of the list of all such objects.

*New in version 2.2.*

**PyInterpreterState**

*PyInterpreterState* Next(PyInterpreterState *interp)

Return the next interpreter state object after *interp* from the list of all such objects.

*New in version 2.2.*

**PyThreadState**

*PyInterpreterState* ThreadHead(PyInterpreterState *interp)

Return the pointer to the first PyThreadState object in the list of threads associated with the interpreter *interp*.

*New in version 2.2.*

**PyThreadState**

*PyThreadState* Next(PyThreadState *tstate)

Return the next thread state object after *tstate* from the list of all such objects belonging to the same PyInterpreterState object.

*New in version 2.2.*
Memory Management
Overview

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the Python memory manager. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if she regularly manipulates object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document.

To avoid memory corruption, extension writers should never try to operate on Python objects with the functions exported by the C library: malloc(), calloc(), realloc() and free(). This will result in
mixed calls between the C allocator and the Python memory manager with fatal consequences, because they implement different algorithms and operate on different heaps. However, one may safely allocate and release memory blocks with the C library allocator for individual purposes, as shown in the following example:

```c
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyString_FromString(buf);
free(buf); /* malloc'ed */
return res;
```

In this example, the memory request for the I/O buffer is handled by the C library allocator. The Python memory manager is involved only in the allocation of the string object returned as a result.

In most situations, however, it is recommended to allocate memory from the Python heap specifically because the latter is under control of the Python memory manager. For example, this is required when the interpreter is extended with new object types written in C. Another reason for using the Python heap is the desire to *inform* the Python memory manager about the memory needs of the extension module. Even when the requested memory is used exclusively for internal, highly-specific purposes, delegating all memory requests to the Python memory manager causes the interpreter to have a more accurate image of its memory footprint as a whole. Consequently, under certain circumstances, the Python memory manager may or may not trigger appropriate actions, like garbage collection, memory compaction or other preventive procedures. Note that by using the C library allocator as shown in the previous example, the allocated memory for the I/O buffer escapes completely the Python memory manager.
Memory Interface

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap:

void* **PyMem_Malloc** (size_t \(n\))

Allocates \(n\) bytes and returns a pointer of type void* to the allocated memory, or NULL if the request fails. Requesting zero bytes returns a distinct non-NULL pointer if possible, as if PyMem_Malloc(1) had been called instead. The memory will not have been initialized in any way.

void* **PyMem_Realloc** (void* \(p\), size_t \(n\))

Resizes the memory block pointed to by \(p\) to \(n\) bytes. The contents will be unchanged to the minimum of the old and the new sizes. If \(p\) is NULL, the call is equivalent to PyMem_Malloc(\(n\)); else if \(n\) is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-NULL. Unless \(p\) is NULL, it must have been returned by a previous call to PyMem_Malloc() or PyMem_Realloc(). If the request fails, PyMem_Realloc() returns NULL and \(p\) remains a valid pointer to the previous memory area.

void **PyMem_Free** (void* \(p\))

Frees the memory block pointed to by \(p\), which must have been returned by a previous call to PyMem_Malloc() or PyMem_Realloc(). Otherwise, or if PyMem_Free(\(p\)) has been called before, undefined behavior occurs. If \(p\) is NULL, no operation is performed.

The following type-oriented macros are provided for convenience. Note that TYPE refers to any C type.

**TYPE*** PyMem_New**(TYPE, size_t \(n\))
Same as \texttt{PyMem\_Malloc()}, but allocates \((n \times \text{sizeof}(\text{TYPE}))\) bytes of memory. Returns a pointer cast to \texttt{TYPE*}. The memory will not have been initialized in any way.

\texttt{TYPE* PyMem\_Resize}(\texttt{void *p, TYPE, size_t n})

Same as \texttt{PyMem\_Realloc()}, but the memory block is resized to \((n \times \text{sizeof}(\text{TYPE}))\) bytes. Returns a pointer cast to \texttt{TYPE*}. On return, \(p\) will be a pointer to the new memory area, or \texttt{NULL} in the event of failure. This is a C preprocessor macro; \(p\) is always reassigned. Save the original value of \(p\) to avoid losing memory when handling errors.

\texttt{void PyMem\_Del}(\texttt{void *p})

Same as \texttt{PyMem\_Free()}.

In addition, the following macro sets are provided for calling the Python memory allocator directly, without involving the C API functions listed above. However, note that their use does not preserve binary compatibility across Python versions and is therefore deprecated in extension modules.

\texttt{PyMem\_MALLOC()}, \texttt{PyMem\_REALLOC()}, \texttt{PyMem\_FREE()}.

\texttt{PyMem\_NEW()}, \texttt{PyMem\_RESIZE()}, \texttt{PyMem\_DEL()}. 
Examples

Here is the example from section Overview, rewritten so that the I/O buffer is allocated from the Python heap by using the first function set:

```c
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyString_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_Malloc */
return res;
```

The same code using the type-oriented function set:

```c
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyString_FromString(buf);
PyMem_Del(buf); /* allocated with PyMem_Malloc */
return res;
```

Note that in the two examples above, the buffer is always manipulated via functions belonging to the same set. Indeed, it is required to use the same memory API family for a given memory block, so that the risk of mixing different allocators is reduced to a minimum. The following code sequence contains two errors, one of which is labeled as fatal because it mixes two different allocators operating on different heaps.

```c
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
```
In addition to the functions aimed at handling raw memory blocks from the Python heap, objects in Python are allocated and released with `PyObject_New()`, `PyObject_NewVar()` and `PyObject_Del()`. These will be explained in the next chapter on defining and implementing new object types in C.
This chapter describes the functions, types, and macros used when defining new object types.

- Allocating Objects on the Heap
- Common Object Structures
- Type Objects
- Number Object Structures
- Mapping Object Structures
- Sequence Object Structures
- Buffer Object Structures
- Supporting Cyclic Garbage Collection
Allocating Objects on the Heap

PyObject* _PyObject_New(PyTypeObject *type)

*Return value: New reference.*

PyVarObject* _PyObject_NewVar(PyTypeObject *type, Py_ssize_t size)

*Return value: New reference.*

*Changed in version 2.5:* This function used an `int` type for `size`. This might require changes in your code for properly supporting 64-bit systems.

void _PyObject_Del(PyObject *op)

PyObject* PyObject_Init(PyObject *op, PyTypeObject *type)

*Return value: Borrowed reference.*

Initialize a newly-allocated object `op` with its type and initial reference. Returns the initialized object. If `type` indicates that the object participates in the cyclic garbage detector, it is added to the detector’s set of observed objects. Other fields of the object are not affected.

PyVarObject* PyObject_InitVar(PyVarObject *op, PyTypeObject *type, Py_ssize_t size)

*Return value: Borrowed reference.*

This does everything `PyObject_Init()` does, and also initializes the length information for a variable-size object.

*Changed in version 2.5:* This function used an `int` type for `size`. This might require changes in your code for properly supporting 64-bit systems.

TYPE* PyObject_New(TYPE, PyTypeObject *type)

*Return value: New reference.*
Allocate a new Python object using the C structure type `TYPE` and the Python type object `type`. Fields not defined by the Python object header are not initialized; the object's reference count will be one. The size of the memory allocation is determined from the `tp_basicsize` field of the type object.

```c
TYPE* PyObject_NewVar(TYPE, PyTypeObject *type, Py_ssize_t size)
```

*Return value: New reference.*

Allocate a new Python object using the C structure type `TYPE` and the Python type object `type`. Fields not defined by the Python object header are not initialized. The allocated memory allows for the `TYPE` structure plus `size` fields of the size given by the `tp_itemsize` field of `type`. This is useful for implementing objects like tuples, which are able to determine their size at construction time. Embedding the array of fields into the same allocation decreases the number of allocations, improving the memory management efficiency.

*Changed in version 2.5:* This function used an `int` type for `size`. This might require changes in your code for properly supporting 64-bit systems.

```c
void PyObject_Del(PyObject *op)
```

Releases memory allocated to an object using `PyObject_New()` or `PyObject_NewVar()`. This is normally called from the `tp_dealloc` handler specified in the object's type. The fields of the object should not be accessed after this call as the memory is no longer a valid Python object.

```c
PyObject* Py_InitModule(char *name, PyMethodDef *methods)
```

*Return value: Borrowed reference.*

Create a new module object based on a name and table of functions, returning the new module object.
Changed in version 2.3: Older versions of Python did not support NULL as the value for the methods argument.

PyObject* Py_InitModule3(char *name, PyMethodDef *methods, char *doc)

Return value: Borrowed reference.
Create a new module object based on a name and table of functions, returning the new module object. If doc is non-NULL, it will be used to define the docstring for the module.

Changed in version 2.3: Older versions of Python did not support NULL as the value for the methods argument.

PyObject* Py_InitModule4(char *name, PyMethodDef *methods, char *doc, PyObject *self, int apiver)

Return value: Borrowed reference.
Create a new module object based on a name and table of functions, returning the new module object. If doc is non-NULL, it will be used to define the docstring for the module. If self is non-NULL, it will be passed to the functions of the module as their (otherwise NULL) first parameter. (This was added as an experimental feature, and there are no known uses in the current version of Python.) For apiver, the only value which should be passed is defined by the constant PYTHON_API_VERSION.

Note: Most uses of this function should probably be using the Py_InitModule3() instead; only use this if you are sure you need it.

Changed in version 2.3: Older versions of Python did not support NULL as the value for the methods argument.

PyObject _Py_NoneStruct
Object which is visible in Python as None. This should only be accessed using the Py_None macro, which evaluates to a pointer
to this object.
Common Object Structures

There are a large number of structures which are used in the definition of object types for Python. This section describes these structures and how they are used.

All Python objects ultimately share a small number of fields at the beginning of the object’s representation in memory. These are represented by the `PyObject` and `PyVarObject` types, which are defined, in turn, by the expansions of some macros also used, whether directly or indirectly, in the definition of all other Python objects.

**PyObject**

All object types are extensions of this type. This is a type which contains the information Python needs to treat a pointer to an object as an object. In a normal “release” build, it contains only the object’s reference count and a pointer to the corresponding type object. It corresponds to the fields defined by the expansion of the `PyObject_HEAD` macro.

**PyVarObject**

This is an extension of `PyObject` that adds the `ob_size` field. This is only used for objects that have some notion of `length`. This type does not often appear in the Python/C API. It corresponds to the fields defined by the expansion of the `PyObject_VAR_HEAD` macro.

These macros are used in the definition of `PyObject` and `PyVarObject`:

**PyObject_HEAD**

This is a macro which expands to the declarations of the fields of the `PyObject` type; it is used when declaring new types which represent objects without a varying length. The specific fields it
expands to depend on the definition of `Py_TRACE_REFS`. By default, that macro is not defined, and `PyObject_HEAD` expands to:

```c
Py_ssize_t ob_refcnt;
PyTypeObject *ob_type;
```

When `Py_TRACE_REFS` is defined, it expands to:

```c
PyObject *ob_next, *ob_prev;
Py_ssize_t ob_refcnt;
PyTypeObject *ob_type;
```

**PyObject_VAR_HEAD**

This is a macro which expands to the declarations of the fields of the `PyVarObject` type; it is used when declaring new types which represent objects with a length that varies from instance to instance. This macro always expands to:

```c
PyObject_HEAD
Py_ssize_t ob_size;
```

Note that `PyObject_HEAD` is part of the expansion, and that its own expansion varies depending on the definition of `Py_TRACE_REFS`.

**Py_TYPE(o)**

This macro is used to access the `ob_type` member of a Python object. It expands to:

```c
(((PyObject*)(o))->ob_type)
```

*New in version 2.6.*

**Py_REFCNT(o)**

This macro is used to access the `ob_refcnt` member of a Python object. It expands to:

```c
(((PyObject*)(o))->ob_refcnt)
```
New in version 2.6.

Py_SIZE(o)
This macro is used to access the ob_size member of a Python object. It expands to:

```c
(((PyVarObject*)(o))->ob_size)
```

New in version 2.6.

PyObject_HEAD_INIT(type)
This is a macro which expands to initialization values for a new PyObject type. This macro expands to:

```c
_PyObject_EXTRA_INIT 1, type,
```

PyVarObject_HEAD_INIT(type, size)
This is a macro which expands to initialization values for a new PyVarObject type, including the ob_size field. This macro expands to:

```c
_PyObject_EXTRA_INIT 1, type, size,
```

PyCFunction
Type of the functions used to implement most Python callables in C. Functions of this type take two PyObject* parameters and return one such value. If the return value is NULL, an exception shall have been set. If not NULL, the return value is interpreted as the return value of the function as exposed in Python. The function must return a new reference.

PyMethodDef
Structure used to describe a method of an extension type. This
structure has four fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>C Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ml_name</td>
<td>char *</td>
<td>name of the method</td>
</tr>
<tr>
<td>ml_meth</td>
<td>PyCFunction</td>
<td>pointer to the C implementation</td>
</tr>
<tr>
<td>ml_flags</td>
<td>int</td>
<td>flag bits indicating how the call should be constructed</td>
</tr>
<tr>
<td>ml_doc</td>
<td>char *</td>
<td>points to the contents of the docstring</td>
</tr>
</tbody>
</table>

The `ml_meth` is a C function pointer. The functions may be of different types, but they always return `PyObject*`. If the function is not of the `PyCFunction`, the compiler will require a cast in the method table. Even though `PyCFunction` defines the first parameter as `PyObject*`, it is common that the method implementation uses the specific C type of the `self` object.

The `ml_flags` field is a bitfield which can include the following flags. The individual flags indicate either a calling convention or a binding convention. Of the calling convention flags, only `METH_VARARGS` and `METH_KEYWORDS` can be combined. Any of the calling convention flags can be combined with a binding flag.

**METH_VARARGS**

This is the typical calling convention, where the methods have the type `PyCFunction`. The function expects two `PyObject*` values. The first one is the `self` object for methods; for module functions, it is the module object. The second parameter (often called `args`) is a tuple object representing all arguments. This parameter is typically processed using `PyArg_ParseTuple()` or `PyArg_UnpackTuple()`.

**METH_KEYWORDS**
Methods with these flags must be of type `PyCFunctionWithKeywords`. The function expects three parameters: `self`, `args`, and a dictionary of all the keyword arguments. The flag is typically combined with `METH_VARARGS`, and the parameters are typically processed using `PyArg_ParseTupleAndKeywords()`.

**METH_NOARGS**
Methods without parameters don’t need to check whether arguments are given if they are listed with the `METH_NOARGS` flag. They need to be of type `PyCFunction`. The first parameter is typically named `self` and will hold a reference to the module or object instance. In all cases the second parameter will be `NULL`.

**METH_O**
Methods with a single object argument can be listed with the `METH_O` flag, instead of invoking `PyArg_ParseTuple()` with a "0" argument. They have the type `PyCFunction`, with the `self` parameter, and a `PyObject*` parameter representing the single argument.

**METH_OLDARGS**
This calling convention is deprecated. The method must be of type `PyCFunction`. The second argument is `NULL` if no arguments are given, a single object if exactly one argument is given, and a tuple of objects if more than one argument is given. There is no way for a function using this convention to distinguish between a call with multiple arguments and a call with a tuple as the only argument.

These two constants are not used to indicate the calling convention but the binding when use with methods of classes. These may not be used for functions defined for modules. At most one of these flags may be set for any given method.
**METH_CLASS**

The method will be passed the type object as the first parameter rather than an instance of the type. This is used to create *class methods*, similar to what is created when using the `classmethod()` built-in function.

*New in version 2.3.*

**METH_STATIC**

The method will be passed `NULL` as the first parameter rather than an instance of the type. This is used to create *static methods*, similar to what is created when using the `staticmethod()` built-in function.

*New in version 2.3.*

One other constant controls whether a method is loaded in place of another definition with the same method name.

**METH_COEXIST**

The method will be loaded in place of existing definitions. Without `METH_COEXIST`, the default is to skip repeated definitions. Since slot wrappers are loaded before the method table, the existence of a `sq_contains` slot, for example, would generate a wrapped method named `__contains__()` and preclude the loading of a corresponding PyCFunction with the same name. With the flag defined, the PyCFunction will be loaded in place of the wrapper object and will co-exist with the slot. This is helpful because calls to PyCFunctions are optimized more than wrapper object calls.

*New in version 2.4.*

**PyMemberDef**

Structure which describes an attribute of a type which corresponds to a C struct member. Its fields are:
<table>
<thead>
<tr>
<th>Field</th>
<th>C Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>char *</td>
<td>name of the member</td>
</tr>
<tr>
<td>type</td>
<td>int</td>
<td>the type of the member in the C struct</td>
</tr>
<tr>
<td>offset</td>
<td>Py_ssize_t</td>
<td>the offset in bytes that the member is located on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the type’s object struct</td>
</tr>
<tr>
<td>flags</td>
<td>int</td>
<td>flag bits indicating if the field should be read-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only or writable</td>
</tr>
<tr>
<td>doc</td>
<td>char *</td>
<td>points to the contents of the docstring</td>
</tr>
</tbody>
</table>

The `type` can be one of many `T_` macros corresponding to various C types. When the member is accessed in Python, it will be converted to the equivalent Python type.

<table>
<thead>
<tr>
<th>Macro name</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_SHORT</td>
<td>short</td>
</tr>
<tr>
<td>T_INT</td>
<td>int</td>
</tr>
<tr>
<td>T_LONG</td>
<td>long</td>
</tr>
<tr>
<td>T_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>T_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>T_STRING</td>
<td>char *</td>
</tr>
<tr>
<td>T_OBJECT</td>
<td>PyObject *</td>
</tr>
<tr>
<td>T_OBJECT_EX</td>
<td>PyObject *</td>
</tr>
<tr>
<td>T_CHAR</td>
<td>char</td>
</tr>
<tr>
<td>T_BYTE</td>
<td>char</td>
</tr>
<tr>
<td>T_UBYTE</td>
<td>unsigned char</td>
</tr>
<tr>
<td>T_UINT</td>
<td>unsigned int</td>
</tr>
<tr>
<td>T_USHORT</td>
<td>unsigned short</td>
</tr>
<tr>
<td>T_ULONG</td>
<td>unsigned long</td>
</tr>
<tr>
<td>T_BOOL</td>
<td>char</td>
</tr>
<tr>
<td>T_LONGLONG</td>
<td>long long</td>
</tr>
<tr>
<td>T_ULONGLONG</td>
<td>unsigned long long</td>
</tr>
</tbody>
</table>
T_PYSSIZET | Py_ssize_t

T_OBJECT and T_OBJECT_EX differ in that T_OBJECT returns None if the member is NULL and T_OBJECT_EX raises an AttributeError. Try to use T_OBJECT_EX over T_OBJECT because T_OBJECT_EX handles use of the `del` statement on that attribute more correctly than T_OBJECT.

flags can be 0 for write and read access or READONLY for read-only access. Using T_STRING for type implies READONLY. Only T_OBJECT and T_OBJECT_EX members can be deleted. (They are set to NULL).

```
PyObject* Py_FindMethod(PyMethodDef table[], PyObject *ob, char *name)
```


Return a bound method object for an extension type implemented in C. This can be useful in the implementation of a `tp_getattro` or `tp_getattr` handler that does not use the `PyObject_GenericGetAttr()` function.
Type Objects

Perhaps one of the most important structures of the Python object system is the structure that defines a new type: the `PyTypeObject` structure. Type objects can be handled using any of the `PyObject_*()` or `PyType_*()` functions, but do not offer much that’s interesting to most Python applications. These objects are fundamental to how objects behave, so they are very important to the interpreter itself and to any extension module that implements new types.

Type objects are fairly large compared to most of the standard types. The reason for the size is that each type object stores a large number of values, mostly C function pointers, each of which implements a small part of the type’s functionality. The fields of the type object are examined in detail in this section. The fields will be described in the order in which they occur in the structure.

Typedefs: `unaryfunc`, `binaryfunc`, `ternaryfunc`, `inquiry`, `coercion`, `intargfunc`, `intintargfunc`, `intobjargproc`, `intintobjargproc`, `objobjargproc`, `destructor`, `freefunc`, `printfunc`, `getattrfunc`, `getattrofunc`, `setattrfunc`, `setattrofunc`, `cmpfunc`, `reprfunc`, `hashfunc`

The structure definition for `PyTypeObject` can be found in `Include/object.h`. For convenience of reference, this repeats the definition found there:

```c
typedef struct _typeobject {
    PyObject_VAR_HEAD
    char *tp_name; /* For printing, in format "<module>.<name>" */
    int tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    printfunc tp_print;
    getattrfunc tp_getattr;
}
```
setattrfunc tp_setattr;
cmpfunc tp_compare;
reprfunc tp_repr;

/* Method suites for standard classes */
PyNumberMethods *tp_as_number;
PySequenceMethods *tp_as_sequence;
PyMappingMethods *tp_as_mapping;

/* More standard operations (here for binary compatibility) */
hashfunc tp_hash;
ternaryfunc tp_call;
reprfunc tp_str;
getattrofunc tp_getattro;
setattrofunc tp_setattro;

/* Functions to access object as input/output buffer */
PyBufferProcs *tp_as_buffer;

/* Flags to define presence of optional/expanded features */
long tp_flags;

char *tp_doc; /* Documentation string */

/* Assigned meaning in release 2.0 */
/* call function for all accessible objects */
traverseproc tp_traverse;

/* delete references to contained objects */
inquiry tp_clear;

/* Assigned meaning in release 2.1 */
/* rich comparisons */
richcmpfunc tp_richcompare;

/* weak reference enabler */
long tp_weaklistoffset;

/* Added in release 2.2 */
/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;

/* Attribute descriptor and subclassing stuff */
struct PyMethodDef *tp_methods;
The type object structure extends the `PyVarObject` structure. The `ob_size` field is used for dynamic types (created by `type_new()`, usually called from a class statement). Note that `PyType_Type` (the metatype) initializes `tp_itemsize`, which means that its instances (i.e. type objects) must have the `ob_size` field.

`PyObject* PyObject._ob_next`  
`PyObject* PyObject._ob_prev`  
These fields are only present when the macro `PY_TRACE_REFS` is defined. Their initialization to `NULL` is taken care of by the `PyObject_HEAD_INIT` macro. For statically allocated objects, these fields always remain `NULL`. For dynamically allocated objects, these two fields are used to link the object into a doubly-linked list of all live objects on the heap. This could be used for various debugging purposes; currently the only use is to print the objects that are still alive at the end of a run when the environment variable `PYTHONDUMPREFS` is set.
These fields are not inherited by subtypes.

Py_ssize_t **PyObject**.ob_refcnt
This is the type object's reference count, initialized to 1 by the **PyObject**.HEAD_INIT macro. Note that for statically allocated type objects, the type's instances (objects whose **ob_type** points back to the type) do not count as references. But for dynamically allocated type objects, the instances do count as references.

This field is not inherited by subtypes.

*Changed in version 2.5:* This field used to be an **int** type. This might require changes in your code for properly supporting 64-bit systems.

PyTypeObject* **PyObject**.ob_type
This is the type's type, in other words its metatype. It is initialized by the argument to the **PyObject**.HEAD_INIT macro, and its value should normally be &PyType_Type. However, for dynamically loadable extension modules that must be usable on Windows (at least), the compiler complains that this is not a valid initializer. Therefore, the convention is to pass **NULL** to the **PyObject**.HEAD_INIT macro and to initialize this field explicitly at the start of the module's initialization function, before doing anything else. This is typically done like this:

```c
Foo_Type.ob_type = &PyType_Type;
```

This should be done before any instances of the type are created. **PyType_Ready()** checks if **ob_type** is **NULL**, and if so, initializes it: in Python 2.2, it is set to &PyType_Type; in Python 2.2.1 and later it is initialized to the **ob_type** field of the base class. **PyType_Ready()** will not change this field if it is non-zero.
In Python 2.2, this field is not inherited by subtypes. In 2.2.1, and in 2.3 and beyond, it is inherited by subtypes.

Py_ssize_t PyVarObject.ob_size
For statically allocated type objects, this should be initialized to zero. For dynamically allocated type objects, this field has a special internal meaning.

This field is not inherited by subtypes.

char* PyTypeObject.tp_name
Pointer to a NUL-terminated string containing the name of the type. For types that are accessible as module globals, the string should be the full module name, followed by a dot, followed by the type name; for built-in types, it should be just the type name. If the module is a submodule of a package, the full package name is part of the full module name. For example, a type named T defined in module M in subpackage Q in package P should have the tp_name initializer "P.Q.M.T".

For dynamically allocated type objects, this should just be the type name, and the module name explicitly stored in the type dict as the value for key '__module__'.

For statically allocated type objects, the tp_name field should contain a dot. Everything before the last dot is made accessible as the __module__ attribute, and everything after the last dot is made accessible as the __name__ attribute.

If no dot is present, the entire tp_name field is made accessible as the __name__ attribute, and the __module__ attribute is undefined (unless explicitly set in the dictionary, as explained above). This means your type will be impossible to pickle. Additionally, it will not be listed in module documentations created with pydoc.
This field is not inherited by subtypes.

Py_ssize_t PyTypeObject.tp_basicsize
Py_ssize_t PyTypeObject.tp_itemsize

These fields allow calculating the size in bytes of instances of the type.

There are two kinds of types: types with fixed-length instances have a zero \texttt{tp\_itemsize} field, types with variable-length instances have a non-zero \texttt{tp\_itemsize} field. For a type with fixed-length instances, all instances have the same size, given in \texttt{tp\_basicsize}.

For a type with variable-length instances, the instances must have an \texttt{ob\_size} field, and the instance size is \texttt{tp\_basicsize} plus \( N \) times \texttt{tp\_itemsize}, where \( N \) is the “length” of the object. The value of \( N \) is typically stored in the instance’s \texttt{ob\_size} field. There are exceptions: for example, long ints use a negative \texttt{ob\_size} to indicate a negative number, and \( N \) is \texttt{abs(ob\_size)} there. Also, the presence of an \texttt{ob\_size} field in the instance layout doesn’t mean that the instance structure is variable-length (for example, the structure for the list type has fixed-length instances, yet those instances have a meaningful \texttt{ob\_size} field).

The basic size includes the fields in the instance declared by the macro \texttt{PyObject\_HEAD} or \texttt{PyObject\_VAR\_HEAD} (whichever is used to declare the instance struct) and this in turn includes the \_ob\_prev and \_ob\_next fields if they are present. This means that the only correct way to get an initializer for the \texttt{tp\_basicsize} is to use the \texttt{sizeof} operator on the struct used to declare the instance layout. The basic size does not include the GC header size (this is new in Python 2.2; in 2.1 and 2.0, the GC header size was included in \texttt{tp\_basicsize}).
These fields are inherited separately by subtypes. If the base type has a non-zero \texttt{tp_itemsize}, it is generally not safe to set \texttt{tp_itemsize} to a different non-zero value in a subtype (though this depends on the implementation of the base type).

A note about alignment: if the variable items require a particular alignment, this should be taken care of by the value of \texttt{tp_basicsize}. Example: suppose a type implements an array of \texttt{double}. \texttt{tp_itemsize} is \texttt{sizeof(double)}. It is the programmer's responsibility that \texttt{tp_basicsize} is a multiple of \texttt{sizeof(double)} (assuming this is the alignment requirement for \texttt{double}).

\textbf{destructor \texttt{PyTypeObject.tp_dealloc}}

A pointer to the instance destructor function. This function must be defined unless the type guarantees that its instances will never be deallocated (as is the case for the singletons \texttt{None} and \texttt{Ellipsis}).

The destructor function is called by the \texttt{Py\_DECREF()} and \texttt{Py\_XDECREF()} macros when the new reference count is zero. At this point, the instance is still in existence, but there are no references to it. The destructor function should free all references which the instance owns, free all memory buffers owned by the instance (using the freeing function corresponding to the allocation function used to allocate the buffer), and finally (as its last action) call the type's \texttt{tp\_free} function. If the type is not subtypable (doesn't have the \texttt{Py\_TPFLAGS\_BASETYPE} flag bit set), it is permissible to call the object deallocator directly instead of via \texttt{tp\_free}. The object deallocator should be the one used to allocate the instance; this is normally \texttt{PyObject\_Del()} if the instance was allocated using \texttt{PyObject\_New()} or \texttt{PyObject\_VarNew()}, or \texttt{PyObject\_GC\_Del()} if the instance was allocated using \texttt{PyObject\_GC\_New()} or \texttt{PyObject\_GC\_NewVar()}. 
This field is inherited by subtypes.

printfunc PyTypeObject.tp_print

An optional pointer to the instance print function.

The print function is only called when the instance is printed to a real file; when it is printed to a pseudo-file (like a StringIO instance), the instance's tp_repr or tp_str function is called to convert it to a string. These are also called when the type's tp_print field is NULL. A type should never implement tp_print in a way that produces different output than tp_repr or tp_str would.

The print function is called with the same signature as PyObject_Print(): int tp_print(PyObject *self, FILE *file, int flags). The self argument is the instance to be printed. The file argument is the stdio file to which it is to be printed. The flags argument is composed of flag bits. The only flag bit currently defined is Py_PRINT_RAW. When the Py_PRINT_RAW flag bit is set, the instance should be printed the same way as tp_str would format it; when the Py_PRINT_RAW flag bit is clear, the instance should be printed the same was as tp_repr would format it. It should return -1 and set an exception condition when an error occurred during the comparison.

It is possible that the tp_print field will be deprecated. In any case, it is recommended not to define tp_print, but instead to rely on tp_repr and tp_str for printing.

This field is inherited by subtypes.

getattrfunc PyTypeObject.tp_getattr

An optional pointer to the get-attribute-string function.
This field is deprecated. When it is defined, it should point to a function that acts the same as the `tp_getattro` function, but taking a C string instead of a Python string object to give the attribute name. The signature is

```
PyObject * tp_getattr(PyObject *o, char *attr_name);
```

This field is inherited by subtypes together with `tp_getattro`: a subtype inherits both `tp_getattr` and `tp_getattro` from its base type when the subtype’s `tp_getattr` and `tp_getattro` are both `NULL`.

`setattrfunc PyTypeObject.tp_setattr`

An optional pointer to the function for setting and deleting attributes.

This field is deprecated. When it is defined, it should point to a function that acts the same as the `tp_setattro` function, but taking a C string instead of a Python string object to give the attribute name. The signature is

```
PyObject * tp_setattr(PyObject *o, char *attr_name, PyObject *v);
```

The `v` argument is set to `NULL` to delete the attribute. This field is inherited by subtypes together with `tp_setattro`: a subtype inherits both `tp_setattr` and `tp_setattro` from its base type when the subtype’s `tp_setattr` and `tp_setattro` are both `NULL`.

`cmpfunc PyTypeObject.tp_compare`

An optional pointer to the three-way comparison function.

The signature is the same as for `PyObject_Compare()`. The function should return `1` if `self` greater than `other`, `0` if `self` is equal to `other`, and `-1` if `self` less than `other`. It should return `-1` and set
an exception condition when an error occurred during the comparison.

This field is inherited by subtypes together with \texttt{tp\_richcompare} and \texttt{tp\_hash}: a subtypes inherits all three of \texttt{tp\_compare}, \texttt{tp\_richcompare}, and \texttt{tp\_hash} when the subtype’s \texttt{tp\_compare}, \texttt{tp\_richcompare}, and \texttt{tp\_hash} are all \texttt{NULL}.

\textbf{reprfunc} \texttt{PyTypeObject.tp\_repr}

An optional pointer to a function that implements the built-in function \texttt{repr()}. The signature is the same as for \texttt{PyObject\_Repr()}; it must return a string or a Unicode object. Ideally, this function should return a string that, when passed to \texttt{eval()}, given a suitable environment, returns an object with the same value. If this is not feasible, it should return a string starting with \texttt{'<'} and ending with \texttt{'>'} from which both the type and the value of the object can be deduced.

When this field is not set, a string of the form \texttt{<\texttt{\%s} object at \texttt{\%p}>} is returned, where \texttt{\%s} is replaced by the type name, and \texttt{\%p} by the object’s memory address.

This field is inherited by subtypes.

\textbf{PyNumberMethods* tp\_as\_number}

Pointer to an additional structure that contains fields relevant only to objects which implement the number protocol. These fields are documented in \texttt{Number Object Structures}.

The \texttt{tp\_as\_number} field is not inherited, but the contained fields are inherited individually.

\textbf{PySequenceMethods* tp\_as\_sequence}

Pointer to an additional structure that contains fields relevant only
to objects which implement the sequence protocol. These fields are documented in \textit{Sequence Object Structures}.

The \texttt{tp\_as\_sequence} field is not inherited, but the contained fields are inherited individually.

\textbf{PyMappingMethods* \texttt{tp\_as\_mapping}}

Pointer to an additional structure that contains fields relevant only to objects which implement the mapping protocol. These fields are documented in \textit{Mapping Object Structures}.

The \texttt{tp\_as\_mapping} field is not inherited, but the contained fields are inherited individually.

\textbf{hashfunc \texttt{PyTypeObject\_tp\_hash}}

An optional pointer to a function that implements the built-in function \texttt{hash()}.

The signature is the same as for \texttt{PyObject\_Hash()}; it must return a C long. The value \texttt{-1} should not be returned as a normal return value; when an error occurs during the computation of the hash value, the function should set an exception and return \texttt{-1}.

This field can be set explicitly to \texttt{PyObject\_HashNotImplemented()} to block inheritance of the hash method from a parent type. This is interpreted as the equivalent of \texttt{\_\_hash\_\_ = None} at the Python level, causing \texttt{isinstance(o, collections.Hashable)} to correctly return \texttt{False}. Note that the converse is also true - setting \texttt{\_\_hash\_\_ = None} on a class at the Python level will result in the \texttt{tp\_hash} slot being set to \texttt{PyObject\_HashNotImplemented()}.

When this field is not set, two possibilities exist: if the \texttt{tp\_compare} and \texttt{tp\_richcompare} fields are both \texttt{NULL}, a default hash value based on the object's address is returned; otherwise, a \texttt{TypeError}
is raised.

This field is inherited by subtypes together with `tp richcompare` and `tp compare`: a subtypes inherits all three of `tp compare`, `tp richcompare`, and `tp hash`, when the subtype's `tp compare`, `tp richcompare` and `tp hash` are all `NULL`.

**ternaryfunc** `PyTypeObject.tp_call`

An optional pointer to a function that implements calling the object. This should be `NULL` if the object is not callable. The signature is the same as for `PyObject_Call()`.

This field is inherited by subtypes.

**reprfunc** `PyTypeObject.tp_str`

An optional pointer to a function that implements the built-in operation `str()`. (Note that `str` is a type now, and `str()` calls the constructor for that type. This constructor calls `PyObject_Str()` to do the actual work, and `PyObject_Str()` will call this handler.)

The signature is the same as for `PyObject_Str()`; it must return a string or a Unicode object. This function should return a “friendly” string representation of the object, as this is the representation that will be used by the print statement.

When this field is not set, `PyObject_Repr()` is called to return a string representation.

This field is inherited by subtypes.

**getattrofunc** `PyTypeObject.tp_getattro`

An optional pointer to the get-attribute function.

The signature is the same as for `PyObject_GetAttr()`. It is usually convenient to set this field to `PyObject_GenericGetAttr()`, which
implements the normal way of looking for object attributes.

This field is inherited by subtypes together with `tp_getattr`: a subtype inherits both `tp_getattr` and `tp_getattro` from its base type when the subtype's `tp_getattr` and `tp_getattro` are both `NULL`.

`setattrofunc PyTypeObject.tp_setattro`

An optional pointer to the function for setting and deleting attributes.

The signature is the same as for `PyObject_SetAttr()`, but setting `v` to `NULL` to delete an attribute must be supported. It is usually convenient to set this field to `PyObject_GenericSetAttr()`, which implements the normal way of setting object attributes.

This field is inherited by subtypes together with `tp_setattr`: a subtype inherits both `tp_setattr` and `tp_setattro` from its base type when the subtype's `tp_setattr` and `tp_setattro` are both `NULL`.

`PyBufferProcs* PyTypeObject.tp_as_buffer`

Pointer to an additional structure that contains fields relevant only to objects which implement the buffer interface. These fields are documented in Buffer Object Structures.

The `tp_as_buffer` field is not inherited, but the contained fields are inherited individually.

`long PyTypeObject.tp_flags`

This field is a bit mask of various flags. Some flags indicate variant semantics for certain situations; others are used to indicate that certain fields in the type object (or in the extension structures referenced via `tp_as_number`, `tp_as_sequence`,
tp_as_mapping, and tp_as_buffer) that were historically not always present are valid; if such a flag bit is clear, the type fields it guards must not be accessed and must be considered to have a zero or NULL value instead.

Inheritance of this field is complicated. Most flag bits are inherited individually, i.e. if the base type has a flag bit set, the subtype inherits this flag bit. The flag bits that pertain to extension structures are strictly inherited if the extension structure is inherited, i.e. the base type's value of the flag bit is copied into the subtype together with a pointer to the extension structure. The Py_TPFLAGS_HAVE_GC flag bit is inherited together with the tp_traverse and tp_clear fields, i.e. if the Py_TPFLAGS_HAVE_GC flag bit is clear in the subtype and the tp_traverse and tp_clear fields in the subtype exist (as indicated by the Py_TPFLAGS_HAVE_RICHCOMPARE flag bit) and have NULL values.

The following bit masks are currently defined; these can be ORed together using the | operator to form the value of the tp_flags field. The macro PyType_HasFeature() takes a type and a flags value, tp and f, and checks whether tp->tp_flags & f is non-zero.

**Py_TPFLAGS_HAVE_GETCHARBUFFER**
If this bit is set, the PyBufferProcs struct referenced by tp_as_buffer has the bf_getcharbuffer field.

**Py_TPFLAGS_HAVE_SEQUENCE_IN**
If this bit is set, the PySequenceMethods struct referenced by tp_as_sequence has the sq_contains field.

**Py_TPFLAGS_GC**
This bit is obsolete. The bit it used to name is no longer in use. The symbol is now defined as zero.
Py_TPFLAGS_HAVE_INPLACEOPS
If this bit is set, the PySequenceMethods struct referenced by tp_as_sequence and the PyNumberMethods structure referenced by tp_as_number contain the fields for in-place operators. In particular, this means that the PyNumberMethods structure has the fields nb_inplace_add, nb_inplace_subtract, nb_inplace_multiply, nb_inplace_divide, nb_inplace_remainder, nb_inplace_power, nb_inplace_lshift, nb_inplace_rshift, nb_inplace_and, nb_inplace_xor, and nb_inplace_or; and the PySequenceMethods struct has the fields sq_inplace_concat and sq_inplace_repeat.

Py_TPFLAGS_CHECKTYPES
If this bit is set, the binary and ternary operations in the PyNumberMethods structure referenced by tp_as_number accept arguments of arbitrary object types, and do their own type conversions if needed. If this bit is clear, those operations require that all arguments have the current type as their type, and the caller is supposed to perform a coercion operation first. This applies to nb_add, nb_subtract, nb_multiply, nb_divide, nb_remainder, nb_divmod, nb_power, nb_lshift, nb_rshift, nb_and, nb_xor, and nb_or.

Py_TPFLAGS_HAVE_RICHCOMPARE
If this bit is set, the type object has the tp_richcompare field, as well as the tp_traverse and the tp_clear fields.

Py_TPFLAGS_HAVE_WEAKREFS
If this bit is set, the tp_weaklistoffset field is defined. Instances of a type are weakly referenceable if the type's tp_weaklistoffset field has a value greater than zero.

Py_TPFLAGS_HAVE_ITER
If this bit is set, the type object has the \texttt{tp_iter} and \texttt{tp_iternext} fields.

\textbf{Py\_TPFLAGS\_HAVE\_CLASS}

If this bit is set, the type object has several new fields defined starting in Python 2.2: \texttt{tp\_methods, tp\_members, tp\_getset, tp\_base, tp\_dict, tp\_descr\_get, tp\_descr\_set, tp\_dictoffset, tp\_init, tp\_alloc, tp\_new, tp\_free, tp\_is\_gc, tp\_bases, tp\_mro, tp\_cache, tp\_subclasses, and tp\_weaklist}.

\textbf{Py\_TPFLAGS\_HEAPTYPE}

This bit is set when the type object itself is allocated on the heap. In this case, the \texttt{ob\_type} field of its instances is considered a reference to the type, and the type object is INCREF’ed when a new instance is created, and DECREF’ed when an instance is destroyed (this does not apply to instances of subtypes; only the type referenced by the instance’s \texttt{ob\_type} gets INCREF’ed or DECREF’ed).

\textbf{Py\_TPFLAGS\_BASETYPE}

This bit is set when the type can be used as the base type of another type. If this bit is clear, the type cannot be subtyped (similar to a “final” class in Java).

\textbf{Py\_TPFLAGS\_READY}

This bit is set when the type object has been fully initialized by \texttt{ PyTuple\_Ready()}.

\textbf{Py\_TPFLAGS\_REAMDYING}

This bit is set while \texttt{ PyTuple\_Ready()} is in the process of initializing the type object.

\textbf{Py\_TPFLAGS\_HAVE\_GC}

This bit is set when the object supports garbage collection. If this bit is set, instances must be created using
PyObject_GC_New() and destroyed using PyObject_GC_Del(). More information in section Supporting Cyclic Garbage Collection. This bit also implies that the GC-related fields tp_traverse and tp_clear are present in the type object; but those fields also exist when Py_TPFLAGS_HAVE_GC is clear but Py_TPFLAGS_HAVE_RICHCOMPARE is set.

Py_TPFLAGS_DEFAULT
This is a bitmask of all the bits that pertain to the existence of certain fields in the type object and its extension structures. Currently, it includes the following bits: Py_TPFLAGS_HAVE_GETCHARBUFFER, Py_TPFLAGS_HAVE_SEQUENCE_IN, Py_TPFLAGS_HAVE_INPLACEOPS, Py_TPFLAGS_HAVE_RICHCOMPARE, Py_TPFLAGS_HAVE_WEAKREFS, Py_TPFLAGS_HAVE_ITER, and Py_TPFLAGS_HAVE_CLASS.

char* PyTypeObject.tp_doc
An optional pointer to a NUL-terminated C string giving the docstring for this type object. This is exposed as the __doc__ attribute on the type and instances of the type.

This field is not inherited by subtypes.

The following three fields only exist if the Py_TPFLAGS_HAVE_RICHCOMPARE flag bit is set.

traverseproc PyTypeObject.tp_traverse
An optional pointer to a traversal function for the garbage collector. This is only used if the Py_TPFLAGS_HAVE_GC flag bit is set. More information about Python’s garbage collection scheme can be found in section Supporting Cyclic Garbage Collection.

The tp_traverse pointer is used by the garbage collector to detect reference cycles. A typical implementation of a
tp_traverse function simply calls Py_VISIT() on each of the instance's members that are Python objects. For example, this is function local_traverse() from the thread extension module:

```c
static int
local_traverse(localobject *self, visitproc visit, void *arg
{
    Py_VISIT(self->args);
    Py_VISIT(self->kw);
    Py_VISIT(self->dict);
    return 0;
}
```

Note that Py_VISIT() is called only on those members that can participate in reference cycles. Although there is also a self->key member, it can only be NULL or a Python string and therefore cannot be part of a reference cycle.

On the other hand, even if you know a member can never be part of a cycle, as a debugging aid you may want to visit it anyway just so the gc module's get_referents() function will include it.

Note that Py_VISIT() requires the visit and arg parameters to local_traverse() to have these specific names; don't name them just anything.

This field is inherited by subtypes together with tp_clear and the Py_TPFLAGS_HAVE_GC flag bit: the flag bit, tp_traverse, and tp_clear are all inherited from the base type if they are all zero in the subtype and the subtype has the Py_TPFLAGS_HAVE_RICHCOMPARE flag bit set.

inquiry PyTypeObject.tp_clear

An optional pointer to a clear function for the garbage collector. This is only used if the Py_TPFLAGS_HAVE_GC flag bit is set.
The `tp_clear` member function is used to break reference cycles in cyclic garbage detected by the garbage collector. Taken together, all `tp_clear` functions in the system must combine to break all reference cycles. This is subtle, and if in any doubt supply a `tp_clear` function. For example, the tuple type does not implement a `tp_clear` function, because it's possible to prove that no reference cycle can be composed entirely of tuples. Therefore the `tp_clear` functions of other types must be sufficient to break any cycle containing a tuple. This isn't immediately obvious, and there's rarely a good reason to avoid implementing `tp_clear`.

Implementations of `tp_clear` should drop the instance's references to those of its members that may be Python objects, and set its pointers to those members to `NULL`, as in the following example:

```c
static int
local_clear(localobject *self)
{
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
    return 0;
}
```

The `Py_CLEAR()` macro should be used, because clearing references is delicate: the reference to the contained object must not be decremented until after the pointer to the contained object is set to `NULL`. This is because decrementing the reference count may cause the contained object to become trash, triggering a chain of reclamation activity that may include invoking arbitrary Python code (due to finalizers, or weakref callbacks, associated with the contained object). If it's possible for such code to reference `self` again, it's important that the pointer to the
contained object be \textit{NULL} at that time, so that \textit{self} knows the contained object can no longer be used. The \texttt{Py\_CLEAR()} macro performs the operations in a safe order.

Because the goal of \texttt{tp\_clear} functions is to break reference cycles, it's not necessary to clear contained objects like Python strings or Python integers, which can't participate in reference cycles. On the other hand, it may be convenient to clear all contained Python objects, and write the type's \texttt{tp\_dealloc} function to invoke \texttt{tp\_clear}.

More information about Python's garbage collection scheme can be found in section \textit{Supporting Cyclic Garbage Collection}.

This field is inherited by subtypes together with \texttt{tp\_traverse} and the \texttt{Py\_TPFLAGS\_HAVE\_GC} flag bit: the flag bit, \texttt{tp\_traverse}, and \texttt{tp\_clear} are all inherited from the base type if they are all zero in the subtype \textit{and} the subtype has the \texttt{Py\_TPFLAGS\_HAVE\_RICHCOMPARE} flag bit set.

\texttt{richcmpfunc} \texttt{PyTypeObject.tp\_richcompare}

An optional pointer to the rich comparison function, whose signature is \texttt{PyObject *tp\_richcompare(PyObject *a, PyObject *b, int op)}.

The function should return the result of the comparison (usually \texttt{Py\_True} or \texttt{Py\_False}). If the comparison is undefined, it must return \texttt{Py\_NotImplemented}, if another error occurred it must return \texttt{NULL} and set an exception condition.

\textbf{Note:} If you want to implement a type for which only a limited set of comparisons makes sense (e.g. \texttt{==} and \texttt{!=}, but not \texttt{<} and friends), directly raise \texttt{TypeError} in the rich comparison
This field is inherited by subtypes together with `tp_compare` and `tp_hash`: a subtype inherits all three of `tp_compare`, `tp_richcompare`, and `tp_hash`, when the subtype’s `tp_compare`, `tp_richcompare`, and `tp_hash` are all `NULL`.

The following constants are defined to be used as the third argument for `tp_richcompare` and for `PyObject_RichCompare()`:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Py_LT</td>
<td>&lt;</td>
</tr>
<tr>
<td>Py_LE</td>
<td>&lt;=</td>
</tr>
<tr>
<td>Py_EQ</td>
<td>==</td>
</tr>
<tr>
<td>Py_NE</td>
<td>!=</td>
</tr>
<tr>
<td>Py_GT</td>
<td>&gt;</td>
</tr>
<tr>
<td>Py_GE</td>
<td>&gt;=</td>
</tr>
</tbody>
</table>

The next field only exists if the `Py_TPFLAGS_HAVE_WEAKREFS` flag bit is set.

`long PyTypeObject.tp_weaklistoffset`

If the instances of this type are weakly referenceable, this field is greater than zero and contains the offset in the instance structure of the weak reference list head (ignoring the GC header, if present); this offset is used by `PyObject_ClearWeakRefs()` and the `PyWeakref_*()` functions. The instance structure needs to include a field of type `PyObject*` which is initialized to `NULL`.

Do not confuse this field with `tp_weaklist`; that is the list head for weak references to the type object itself.

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype uses a different weak reference list head than the base type.
Since the list head is always found via `tp_weaklistoffset`, this should not be a problem.

When a type defined by a class statement has no `__slots__` declaration, and none of its base types are weakly referenceable, the type is made weakly referenceable by adding a weak reference list head slot to the instance layout and setting the `tp_weaklistoffset` of that slot's offset.

When a type's `__slots__` declaration contains a slot named `__weakref__`, that slot becomes the weak reference list head for instances of the type, and the slot's offset is stored in the type's `tp_weaklistoffset`.

When a type's `__slots__` declaration does not contain a slot named `__weakref__`, the type inherits its `tp_weaklistoffset` from its base type.

The next two fields only exist if the `Py_TPFLAGS_HAVE_ITER` flag bit is set.

**getiterfunc** `PyTypeObject.tp_iter`

An optional pointer to a function that returns an iterator for the object. Its presence normally signals that the instances of this type are iterable (although sequences may be iterable without this function, and classic instances always have this function, even if they don't define an `__iter__()` method).

This function has the same signature as `PyObject_GetIter()`.

This field is inherited by subtypes.

**iternextfunc** `PyTypeObject.tp_iternext`

An optional pointer to a function that returns the next item in an iterator. When the iterator is exhausted, it must return `NULL`; a
StopIteration exception may or may not be set. When another error occurs, it must return NULL too. Its presence normally signals that the instances of this type are iterators (although classic instances always have this function, even if they don’t define a next() method).

Iterator types should also define the tp_iter function, and that function should return the iterator instance itself (not a new iterator instance).

This function has the same signature as PyIter_Next().

This field is inherited by subtypes.

The next fields, up to and including tp_weaklist, only exist if the Py_TPFLAGS_HAVE_CLASS flag bit is set.

struct PyMethodDef* PyTypeObject.tp_methods
An optional pointer to a static NULL-terminated array of PyMethodDef structures, declaring regular methods of this type.

For each entry in the array, an entry is added to the type's dictionary (see tp_dict below) containing a method descriptor.

This field is not inherited by subtypes (methods are inherited through a different mechanism).

struct PyMemberDef* PyTypeObject.tp_members
An optional pointer to a static NULL-terminated array of PyMemberDef structures, declaring regular data members (fields or slots) of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see tp_dict below) containing a member descriptor.
This field is not inherited by subtypes (members are inherited through a different mechanism).

**struct PyGetSetDef** *PyTypeObject.tp_getset*

An optional pointer to a static NULL-terminated array of *PyGetSetDef* structures, declaring computed attributes of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a getset descriptor.

This field is not inherited by subtypes (computed attributes are inherited through a different mechanism).

Docs for *PyGetSetDef*:

```c
typedef PyObject *(getter)(PyObject *, void *);
typedef int *(setter)(PyObject *, PyObject *, void *);
typedef struct PyGetSetDef {
    char *name;       /* attribute name */
    getter get;      /* C function to get the attribute */
    setter set;      /* C function to set or delete the attribute */
    char *doc;       /* optional doc string */
    void *closure;   /* optional additional data for getter and setter */
} PyGetSetDef;
```

**PyTypeObject** *PyTypeObject.tp_base*

An optional pointer to a base type from which type properties are inherited. At this level, only single inheritance is supported; multiple inheritance require dynamically creating a type object by calling the metatype.

This field is not inherited by subtypes (obviously), but it defaults to &PyBaseObject_Type (which to Python programmers is known as the type *object*).
PyObject* `PyTypeObject.tp_dict`

The type’s dictionary is stored here by `PyType_Ready()`.

This field should normally be initialized to NULL before `PyType_Ready` is called; it may also be initialized to a dictionary containing initial attributes for the type. Once `PyType_Ready()` has initialized the type, extra attributes for the type may be added to this dictionary only if they don’t correspond to overloaded operations (like `__add__()`).

This field is not inherited by subtypes (though the attributes defined in here are inherited through a different mechanism).

descrgetfunc `PyTypeObject.tp_descr_get`

An optional pointer to a “descriptor get” function.

The function signature is

```c
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObj
```

This field is inherited by subtypes.

descrsetfunc `PyTypeObject.tp_descr_set`

An optional pointer to a function for setting and deleting a descriptor’s value.

The function signature is

```c
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *va
```

The `value` argument is set to NULL to delete the value. This field is inherited by subtypes.

long `PyTypeObject.tp_dictoffset`

If the instances of this type have a dictionary containing instance
variables, this field is non-zero and contains the offset in the instances of the type of the instance variable dictionary; this offset is used by `PyObject_GenericGetAttr()`.

Do not confuse this field with `tp_dict`; that is the dictionary for attributes of the type object itself.

If the value of this field is greater than zero, it specifies the offset from the start of the instance structure. If the value is less than zero, it specifies the offset from the end of the instance structure. A negative offset is more expensive to use, and should only be used when the instance structure contains a variable-length part. This is used for example to add an instance variable dictionary to subtypes of `str` or `tuple`. Note that the `tp_basicsize` field should account for the dictionary added to the end in that case, even though the dictionary is not included in the basic object layout. On a system with a pointer size of 4 bytes, `tp_dictoffset` should be set to -4 to indicate that the dictionary is at the very end of the structure.

The real dictionary offset in an instance can be computed from a negative `tp_dictoffset` as follows:

```python
dictoffset = tp_basicsize + abs(ob_size)*tp_itemsize + tp_dictoffset
if dictoffset is not aligned on sizeof(void*):
    round up to sizeof(void*)
```

where `tp_basicsize`, `tp_itemsize` and `tp_dictoffset` are taken from the type object, and `ob_size` is taken from the instance. The absolute value is taken because long ints use the sign of `ob_size` to store the sign of the number. (There's never a need to do this calculation yourself; it is done for you by `_PyObject_GetDictPtr()`.)
This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype instances store the dictionary at a difference offset than the base type. Since the dictionary is always found via tp_dictoffset, this should not be a problem.

When a type defined by a class statement has no __slots__ declaration, and none of its base types has an instance variable dictionary, a dictionary slot is added to the instance layout and the tp_dictoffset is set to that slot’s offset.

When a type defined by a class statement has a __slots__ declaration, the type inherits its tp_dictoffset from its base type.

(Adding a slot named __dict__ to the __slots__ declaration does not have the expected effect, it just causes confusion. Maybe this should be added as a feature just like __weakref__ though.)

initproc PyTypeObject.tp_init

An optional pointer to an instance initialization function.

This function corresponds to the __init__() method of classes. Like __init__(), it is possible to create an instance without calling __init__(), and it is possible to reinitialize an instance by calling its __init__() method again.

The function signature is

```c
int tp_init(PyObject *self, PyObject *args, PyObject *kwds)
```

The self argument is the instance to be initialized; the args and kwds arguments represent positional and keyword arguments of the call to __init__().

The tp_init function, if not NULL, is called when an instance is
created normally by calling its type, after the type's \texttt{tp\_new} function has returned an instance of the type. If the \texttt{tp\_new} function returns an instance of some other type that is not a subtype of the original type, no \texttt{tp\_init} function is called; if \texttt{tp\_new} returns an instance of a subtype of the original type, the subtype's \texttt{tp\_init} is called. (VERSION NOTE: described here is what is implemented in Python 2.2.1 and later. In Python 2.2, the \texttt{tp\_init} of the type of the object returned by \texttt{tp\_new} was always called, if not \texttt{NULL}.)

This field is inherited by subtypes.

\textbf{allocfunc} \texttt{PyTypeObject}.'\texttt{tp\_alloc}

An optional pointer to an instance allocation function.

The function signature is

\begin{verbatim}
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t nitems)
\end{verbatim}

The purpose of this function is to separate memory allocation from memory initialization. It should return a pointer to a block of memory of adequate length for the instance, suitably aligned, and initialized to zeros, but with \texttt{ob\_refcnt} set to 1 and \texttt{ob\_type} set to the type argument. If the type's \texttt{tp\_itemsize} is non-zero, the object's \texttt{ob\_size} field should be initialized to \texttt{nitems} and the length of the allocated memory block should be \texttt{tp\_basicsize + nitems*tp\_itemsize}, rounded up to a multiple of \texttt{sizeof(void*)}; otherwise, \texttt{nitems} is not used and the length of the block should be \texttt{tp\_basicsize}.

Do not use this function to do any other instance initialization, not even to allocate additional memory; that should be done by \texttt{tp\_new}. 


This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement); in the latter, this field is always set to `PyType_GenericAlloc()`, to force a standard heap allocation strategy. That is also the recommended value for statically defined types.

newfunc `PyTypeObject.tp_new`

An optional pointer to an instance creation function.

If this function is `NULL` for a particular type, that type cannot be called to create new instances; presumably there is some other way to create instances, like a factory function.

The function signature is

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds)
```

The subtype argument is the type of the object being created; the `args` and `kwds` arguments represent positional and keyword arguments of the call to the type. Note that subtype doesn't have to equal the type whose `tp_new` function is called; it may be a subtype of that type (but not an unrelated type).

The `tp_new` function should call `subtype->tp_alloc(subtype, nitems)` to allocate space for the object, and then do only as much further initialization as is absolutely necessary. Initialization that can safely be ignored or repeated should be placed in the `tp_init` handler. A good rule of thumb is that for immutable types, all initialization should take place in `tp_new`, while for mutable types, most initialization should be deferred to `tp_init`.

This field is inherited by subtypes, except it is not inherited by static types whose `tp_base` is `NULL` or `&PyBaseObject_Type`. The latter exception is a precaution so that old extension types don't
become callable simply by being linked with Python 2.2.

destructor **PyTypeObject.tp_free**

An optional pointer to an instance deallocation function.

The signature of this function has changed slightly: in Python 2.2 and 2.2.1, its signature is *destructor*:

```c
void tp_free(PyObject *)
```

In Python 2.3 and beyond, its signature is *freefunc*:

```c
void tp_free(void *)
```

The only initializer that is compatible with both versions is \_PyObject\_Del, whose definition has suitably adapted in Python 2.3.

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement); in the latter, this field is set to a deallocator suitable to match *PyType\_GenericAlloc()* and the value of the *Py\_TPFLAGS\_HAVE\_GC* flag bit.

**inquiry** **PyTypeObject.tp_is_gc**

An optional pointer to a function called by the garbage collector.

The garbage collector needs to know whether a particular object is collectible or not. Normally, it is sufficient to look at the object's type's *tp_flags* field, and check the *Py\_TPFLAGS\_HAVE\_GC* flag bit. But some types have a mixture of statically and dynamically allocated instances, and the statically allocated instances are not collectible. Such types should define this function; it should return 1 for a collectible instance, and 0 for a non-collectible instance. The signature is
int tp_is_gc(PyObject *self)

(The only example of this are types themselves. The metatype, `PyType_Type`, defines this function to distinguish between statically and dynamically allocated types.)

This field is inherited by subtypes. (VERSION NOTE: in Python 2.2, it was not inherited. It is inherited in 2.2.1 and later versions.)

**PyObject** `PyTypeObject.tp_bases`  
Tuple of base types.

This is set for types created by a class statement. It should be `NULL` for statically defined types.

This field is not inherited.

**PyObject** `PyTypeObject.tp_mro`  
Tuple containing the expanded set of base types, starting with the type itself and ending with `object`, in Method Resolution Order.

This field is not inherited; it is calculated fresh by `PyType_Ready()`.

**PyObject** `PyTypeObject.tp_cache`  
Unused. Not inherited. Internal use only.

**PyObject** `PyTypeObject.tp_subclasses`  
List of weak references to subclasses. Not inherited. Internal use only.

**PyObject** `PyTypeObject.tp_weaklist`  
Weak reference list head, for weak references to this type object. Not inherited. Internal use only.

The remaining fields are only defined if the feature test macro
**COUNT_ALLOCS** is defined, and are for internal use only. They are documented here for completeness. None of these fields are inherited by subtypes.

`Py_ssize_t PyTypeObject.tp_allocs`

Number of allocations.

`Py_ssize_t PyTypeObject.tp_frees`

Number of frees.

`Py_ssize_t PyTypeObject.tp_maxalloc`

Maximum simultaneously allocated objects.

`PyTypeObject* PyTypeObject.tp_next`

Pointer to the next type object with a non-zero `tp_allocs` field.

Also, note that, in a garbage collected Python, `tp_dealloc` may be called from any Python thread, not just the thread which created the object (if the object becomes part of a refcount cycle, that cycle might be collected by a garbage collection on any thread). This is not a problem for Python API calls, since the thread on which `tp_dealloc` is called will own the Global Interpreter Lock (GIL). However, if the object being destroyed in turn destroys objects from some other C or C++ library, care should be taken to ensure that destroying those objects on the thread which called `tp_dealloc` will not violate any assumptions of the library.
Number Object Structures

PyNumberMethods

This structure holds pointers to the functions which an object uses to implement the number protocol. Almost every function below is used by the function of similar name documented in the Number Protocol section.

Here is the structure definition:

typedef struct {
    binaryfunc nb_add;
    binaryfunc nb_subtract;
    binaryfunc nb_multiply;
    binaryfunc nb_divide;
    binaryfunc nb_remainder;
    binaryfunc nb_divmod;
    ternaryfunc nb_power;
    unaryfunc nb_negative;
    unaryfunc nb_positive;
    unaryfunc nb_absolute;
    inquiry nb_nonzero; /* Used by PyObject_IsTrue */
    unaryfunc nb_invert;
    binaryfunc nb_lshift;
    binaryfunc nb_rshift;
    binaryfunc nb_and;
    binaryfunc nb_xor;
    binaryfunc nb_or;
    coercion nb_coerce; /* Used by the coerce() function */
    unaryfunc nb_int;
    unaryfunc nb_long;
    unaryfunc nb_float;
    unaryfunc nb_oct;
    unaryfunc nb_hex;

    /* Added in release 2.0 */
    binaryfunc nb_inplace_add;
    binaryfunc nb_inplace_subtract;
    binaryfunc nb_inplace_multiply;
    binaryfunc nb_inplace_divide;
    binaryfunc nb_inplace_remainder;
    ternaryfunc nb_inplace_power;
}
Binary and ternary functions may receive different kinds of arguments, depending on the flag bit `Py_TPFLAGS_CHECKTYPES`:

- If `Py_TPFLAGS_CHECKTYPES` is not set, the function arguments are guaranteed to be of the object’s type; the caller is responsible for calling the coercion method specified by the `nb_coerce` member to convert the arguments:

  coercion `PyNumberMethods.nb_coerce`

  This function is used by `PyNumber_CoerceEx()` and has the same signature. The first argument is always a pointer to an object of the defined type. If the conversion to a common “larger” type is possible, the function replaces the pointers with new references to the converted objects and returns 0. If the conversion is not possible, the function returns 1. If an error condition is set, it will return -1.

- If the `Py_TPFLAGS_CHECKTYPES` flag is set, binary and ternary functions must check the type of all their operands, and implement the necessary conversions (at least one of the operands is an instance of the defined type). This is the
recommended way; with Python 3 coercion will disappear completely.

If the operation is not defined for the given operands, binary and ternary functions must return `Py_NotImplemented`, if another error occurred they must return `NULL` and set an exception.
Mapping Object Structures

**PyMappingMethods**
This structure holds pointers to the functions which an object uses to implement the mapping protocol. It has three members:

lenfunc **PyMappingMethods.mp_length**
This function is used by `PyMapping_Length()` and `PyObject_Size()`, and has the same signature. This slot may be set to `NULL` if the object has no defined length.

binaryfunc **PyMappingMethods.mp_subscript**
This function is used by `PyObject_GetItem()` and has the same signature. This slot must be filled for the `PyMapping_Check()` function to return 1, it can be `NULL` otherwise.

objobjargproc **PyMappingMethods.mp_ass_subscript**
This function is used by `PyObject_SetItem()` and `PyObject_DelItem()`. It has the same signature as `PyObject_SetItem()`, but `v` can also be set to `NULL` to delete an item. If this slot is `NULL`, the object does not support item assignment and deletion.
**Sequence Object Structures**

**PySequenceMethods**

This structure holds pointers to the functions which an object uses to implement the sequence protocol.

**lenfunc PySequenceMethods.sq_length**

This function is used by `PySequence_Size()` and `PyObject_Size()`, and has the same signature.

**binaryfunc PySequenceMethods.sq_concat**

This function is used by `PySequence_Concat()` and has the same signature. It is also used by the `+` operator, after trying the numeric addition via the `nb_add` slot.

**ssizeargfunc PySequenceMethods.sq_repeat**

This function is used by `PySequence_Repeat()` and has the same signature. It is also used by the `*` operator, after trying numeric multiplication via the `nb_multiply` slot.

**ssizeargfunc PySequenceMethods.sq_item**

This function is used by `PySequence_GetItem()` and has the same signature. This slot must be filled for the `PySequence_Check()` function to return 1, it can be `NULL` otherwise.

Negative indexes are handled as follows: if the `sq_length` slot is filled, it is called and the sequence length is used to compute a positive index which is passed to `sq_item`. If `sq_length` is `NULL`, the index is passed as is to the function.

**ssizeobjargproc PySequenceMethods.sq_ass_item**

This function is used by `PySequence_SetItem()` and has the same signature. This slot may be left to `NULL` if the object does not
support item assignment and deletion.

**objobjproc** `PySequenceMethods.sq_contains`

This function may be used by `PySequence_Contains()` and has the same signature. This slot may be left to `NULL`, in this case `PySequence_Contains()` simply traverses the sequence until it finds a match.

**binaryfunc** `PySequenceMethods.sq_inplace_concat`

This function is used by `PySequence_InPlaceConcat()` and has the same signature. It should modify its first operand, and return it.

**ssizeargfunc** `PySequenceMethods.sq_inplace_repeat`

This function is used by `PySequence_InPlaceRepeat()` and has the same signature. It should modify its first operand, and return it.
Buffer Object Structures

The buffer interface exports a model where an object can expose its internal data as a set of chunks of data, where each chunk is specified as a pointer/length pair. These chunks are called *segments* and are presumed to be non-contiguous in memory.

If an object does not export the buffer interface, then its `tp_as_buffer` member in the `PyTypeObject` structure should be `NULL`. Otherwise, the `tp_as_buffer` will point to a `PyBufferProcs` structure.

**Note:** It is very important that your `PyTypeObject` structure uses `Py_TPFLAGS_DEFAULT` for the value of the `tp_flags` member rather than 0. This tells the Python runtime that your `PyBufferProcs` structure contains the `bf_getcharbuffer` slot. Older versions of Python did not have this member, so a new Python interpreter using an old extension needs to be able to test for its presence before using it.

**PyBufferProcs**

Structure used to hold the function pointers which define an implementation of the buffer protocol.

The first slot is `bf_getreadbuffer`, of type `getreadbufferproc`. If this slot is `NULL`, then the object does not support reading from the internal data. This is non-sensical, so implementors should fill this in, but callers should test that the slot contains a non-`NULL` value.

The next slot is `bf_getwritebuffer` having type `getwritebufferproc`. This slot may be `NULL` if the object does not allow writing into its returned buffers.
The third slot is `bf_getsegcount`, with type `getsegcountproc`. This slot must not be `NULL` and is used to inform the caller how many segments the object contains. Simple objects such as `PyString_Type` and `PyBuffer_Type` objects contain a single segment.

The last slot is `bf_getcharbuffer`, of type `getcharbufferproc`. This slot will only be present if the `Py_TPFLAGS_HAVE_GETCHARBUFFER` flag is present in the `tp_flags` field of the object’s `PyTypeObject`. Before using this slot, the caller should test whether it is present by using the `PyType_HasFeature()` function. If the flag is present, `bf_getcharbuffer` may be `NULL`, indicating that the object’s contents cannot be used as 8-bit characters. The slot function may also raise an error if the object’s contents cannot be interpreted as 8-bit characters. For example, if the object is an array which is configured to hold floating point values, an exception may be raised if a caller attempts to use `bf_getcharbuffer` to fetch a sequence of 8-bit characters. This notion of exporting the internal buffers as “text” is used to distinguish between objects that are binary in nature, and those which have character-based content.

**Note:** The current policy seems to state that these characters may be multi-byte characters. This implies that a buffer size of `N` does not mean there are `N` characters present.

```c
Py_TPFLAGS_HAVE_GETCHARBUFFER
Flag bit set in the type structure to indicate that the `bf_getcharbuffer` slot is known. This being set does not indicate that the object supports the buffer interface or that the `bf_getcharbuffer` slot is non-NULL.

Py_ssize_t (*readbufferproc)(PyObject *self,
Py_ssize_t segment, void **ptrptr)
```
Return a pointer to a readable segment of the buffer in *ptrptr. This function is allowed to raise an exception, in which case it must return -1. The segment which is specified must be zero or positive, and strictly less than the number of segments returned by the bf_getsegcount slot function. On success, it returns the length of the segment, and sets *ptrptr to a pointer to that memory.

Py_ssize_t (*writebufferproc)(PyObject *self, Py_ssize_t segment, void **ptrptr)

Return a pointer to a writable memory buffer in *ptrptr, and the length of that segment as the function return value. The memory buffer must correspond to buffer segment segment. Must return -1 and set an exception on error. TypeError should be raised if the object only supports read-only buffers, and SystemError should be raised when segment specifies a segment that doesn't exist.

Py_ssize_t (*segcountproc)(PyObject *self, Py_ssize_t *lenp)

Return the number of memory segments which comprise the buffer. If lenp is not NULL, the implementation must report the sum of the sizes (in bytes) of all segments in *lenp. The function cannot fail.

Py_ssize_t (*charbufferproc)(PyObject *self, Py_ssize_t segment, char **ptrptr)

Return the size of the segment segment that ptrptr is set to. *ptrptr is set to the memory buffer. Returns -1 on error.
Supporting Cyclic Garbage Collection

Python's support for detecting and collecting garbage which involves circular references requires support from object types which are “containers” for other objects which may also be containers. Types which do not store references to other objects, or which only store references to atomic types (such as numbers or strings), do not need to provide any explicit support for garbage collection.

To create a container type, the `tp_flags` field of the type object must include the `Py_TPFLAGS_HAVE_GC` and provide an implementation of the `tp_traverse` handler. If instances of the type are mutable, a `tp_clear` implementation must also be provided.

`Py_TPFLAGS_HAVE_GC`

Objects with a type with this flag set must conform with the rules documented here. For convenience these objects will be referred to as container objects.

Constructors for container types must conform to two rules:

1. The memory for the object must be allocated using `PyObject_GC_New()` or `PyObject_GC_NewVar()`.
2. Once all the fields which may contain references to other containers are initialized, it must call `PyObject_GC_Track()`.

`TYPE* PyObject_GC_New(TYPE, PyTypeObject *type)`

Analogous to `PyObject_New()` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

`TYPE* PyObject_GC_NewVar(TYPE, PyTypeObject *type, Py_ssize_t size)`
Analogous to `PyObject_NewVar()` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

Changed in version 2.5: This function used an `int` type for size. This might require changes in your code for properly supporting 64-bit systems.

```c
TYPE* PyObject_GC_Resize(TYPE, PyVarObject *op, Py_ssize_t newsize)
```
Resize an object allocated by `PyObject_NewVar()`. Returns the resized object or `NULL` on failure.

Changed in version 2.5: This function used an `int` type for `newsize`. This might require changes in your code for properly supporting 64-bit systems.

```c
void PyObject_GC_Track(PyObject *op)
```
Adds the object `op` to the set of container objects tracked by the collector. The collector can run at unexpected times so objects must be valid while being tracked. This should be called once all the fields followed by the `tp_traverse` handler become valid, usually near the end of the constructor.

```c
void _PyObject_GC_TRACK(PyObject *op)
```
A macro version of `PyObject_GC_Track()`. It should not be used for extension modules.

Similarly, the deallocator for the object must conform to a similar pair of rules:

1. Before fields which refer to other containers are invalidated, `PyObject_GC_UnTrack()` must be called.
2. The object’s memory must be deallocated using `PyObject_GC_Del()`. 
void **PyObject**_**GC**_**Del**(void *op)

 Releepse memory allocated to an object using **PyObject**_**GC**_**New**(())
or **PyObject**_**GC**_**NewVar**(()).

void **PyObject**_**GC**_**UnTrack**(void *op)

 Remove the object op from the set of container objects tracked by the collector. Note that **PyObject**_**GC**_**Track**(()) can be called again on this object to add it back to the set of tracked objects. The deallocator (**tp**_**dealloc** handler) should call this for the object before any of the fields used by the **tp**_**traverse** handler become invalid.

void **PyObject**_**GC**_**UNTRACK**(PyObject *op)

 A macro version of **PyObject**_**GC**_**UnTrack**(()). It should not be used for extension modules.

The **tp**_**traverse** handler accepts a function parameter of this type:

int (*visitproc)(PyObject *object, void *arg)

 Type of the visitor function passed to the **tp**_**traverse** handler. The function should be called with an object to traverse as object and the third parameter to the **tp**_**traverse** handler as arg. The Python core uses several visitor functions to implement cyclic garbage detection; it's not expected that users will need to write their own visitor functions.

The **tp**_**traverse** handler must have the following type:

int (*traverseproc)(PyObject *self, visitproc visit, void *arg)

 Traversal function for a container object. Implementations must call the visit function for each object directly contained by self, with the parameters to visit being the contained object and the arg value passed to the handler. The visit function must not be called with a **NULL** object argument. If visit returns a non-zero
value that value should be returned immediately.

To simplify writing `tp_traverse` handlers, a `Py_VISIT()` macro is provided. In order to use this macro, the `tp_traverse` implementation must name its arguments exactly `visit` and `arg`:

```c
void Py_VISIT(PyObject *o)
    If o is not NULL, call the visit callback, with arguments o and arg. If visit returns a non-zero value, then return it. Using this macro, `tp_traverse` handlers look like:

```c
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

New in version 2.4.

The `tp_clear` handler must be of the `inquiry` type, or `NULL` if the object is immutable.

```c
int (*inquiry)(PyObject *self)
    Drop references that may have created reference cycles. Immutable objects do not have to define this method since they can never directly create reference cycles. Note that the object must still be valid after calling this method (don’t just call `Py_DECREF()` on a reference). The collector will call this method if it detects that this object is involved in a reference cycle.
```
Porting Python 2 Code to Python 3

author: Brett Cannon

Abstract

With Python 3 being the future of Python while Python 2 is still in active use, it is good to have your project available for both major releases of Python. This guide is meant to help you figure out how best to support both Python 2 & 3 simultaneously.

If you are looking to port an extension module instead of pure Python code, please see Porting Extension Modules to Python 3.

If you would like to read one core Python developer’s take on why Python 3 came into existence, you can read Nick Coghlan's Python 3 Q & A.

For help with porting, you can email the python-porting mailing list with questions.
The Short Explanation

To make your project be single-source Python 2/3 compatible, the basic steps are:

1. Update your code to drop support for Python 2.5 or older (supporting only Python 2.7 is ideal)
2. Make sure you have good test coverage (coverage.py can help; pip install coverage)
3. Learn the differences between Python 2 & 3
4. Use Modernize or Futurize to update your code (pip install modernize or pip install future, respectively)
5. Use Pylint to help make sure you don’t regress on your Python 3 support (if only supporting Python 2.7/3.4 or newer; pip install pylint)
6. Use caniusepython3 to find out which of your dependencies are blocking your use of Python 3 (pip install caniusepython3)
7. Once your dependencies are no longer blocking you, use continuous integration to make sure you stay compatible with Python 2 & 3 (tox can help test against multiple versions of Python; pip install tox)

If you are dropping support for Python 2 entirely, then after you learn the differences between Python 2 & 3 you can run 2to3 over your code and skip the rest of the steps outlined above.
A key point about supporting Python 2 & 3 simultaneously is that you can start **today**! Even if your dependencies are not supporting Python 3 yet that does not mean you can’t modernize your code now to support Python 3. Most changes required to support Python 3 lead to cleaner code using newer practices even in Python 2.

Another key point is that modernizing your Python 2 code to also support Python 3 is largely automated for you. While you might have to make some API decisions thanks to Python 3 clarifying text data versus binary data, the lower-level work is now mostly done for you and thus can at least benefit from the automated changes immediately.

Keep those key points in mind while you read on about the details of porting your code to support Python 2 & 3 simultaneously.

**Drop support for Python 2.5 and older (at least)**

While you can make Python 2.5 work with Python 3, it is much easier if you only have to work with Python 2.6 or newer (and easier still if you only have to work with Python 2.7). If dropping Python 2.5 is not an option then the six project can help you support Python 2.5 & 3 simultaneously (`pip install six`). Do realize, though, that nearly all the projects listed in this HOWTO will not be available to you.

If you are able to only support Python 2.6 or newer, then the required changes to your code should continue to look and feel like idiomatic Python code. At worst you will have to use a function instead of a method in some instances or have to import a function instead of using a built-in one, but otherwise the overall transformation should
not feel foreign to you.

But please aim for Python 2.7. Bugfixes for that version of Python will continue until 2020 while Python 2.6 is no longer supported. There are also some tools mentioned in this HOWTO which do not support Python 2.6 (e.g., Pylint), and this will become more commonplace as time goes on.

Make sure you specify the proper version support in your setup.py file

In your setup.py file you should have the proper trove classifier specifying what versions of Python you support. As your project does not support Python 3 yet you should at least have Programming Language :: Python :: 2 :: Only specified. Ideally you should also specify each major/minor version of Python that you do support, e.g. Programming Language :: Python :: 2.7.

Have good test coverage

Once you have your code supporting the oldest version of Python 2 you want it to, you will want to make sure your test suite has good coverage. A good rule of thumb is that if you want to be confident enough in your test suite that any failures that appear after having tools rewrite your code are actual bugs in the tools and not in your code. If you want a number to aim for, try to get over 80% coverage (and don't feel bad if you can't easily get past 90%). If you don't already have a tool to measure test coverage then coverage.py is recommended.

Learn the differences between Python 2 & 3

Once you have your code well-tested you are ready to begin porting
your code to Python 3! But to fully understand how your code is going to change and what you want to look out for while you code, you will want to learn what changes Python 3 makes in terms of Python 2. Typically the two best ways of doing that is reading the “What’s New” doc for each release of Python 3 and the Porting to Python 3 book (which is free online). There is also a handy cheat sheet from the Python-Future project.

**Update your code**

Once you feel like you know what is different in Python 3 compared to Python 2, it’s time to update your code! You have a choice between two tools in porting your code automatically: Modernize and Futurize. Which tool you choose will depend on how much like Python 3 you want your code to be. Futurize does its best to make Python 3 idioms and practices exist in Python 2, e.g. backporting the bytes type from Python 3 so that you have semantic parity between the major versions of Python. Modernize, on the other hand, is more conservative and targets a Python 2/3 subset of Python, relying on six to help provide compatibility.

Regardless of which tool you choose, they will update your code to run under Python 3 while staying compatible with the version of Python 2 you started with. Depending on how conservative you want to be, you may want to run the tool over your test suite first and visually inspect the diff to make sure the transformation is accurate. After you have transformed your test suite and verified that all the tests still pass as expected, then you can transform your application code knowing that any tests which fail is a translation failure.

Unfortunately the tools can’t automate everything to make your code work under Python 3 and so there are a handful of things you will need to update manually to get full Python 3 support (which of these steps are necessary vary between the tools). Read the
documentation for the tool you choose to use to see what it fixes by default and what it can do optionally to know what will (not) be fixed for you and what you may have to fix on your own (e.g. using `io.open()` over the built-in `open()` function is off by default in Modernize). Luckily, though, there are only a couple of things to watch out for which can be considered large issues that may be hard to debug if not watched for.

**Division**

In Python 3, `5 / 2 == 2.5` and not 2; all division between `int` values result in a `float`. This change has actually been planned since Python 2.2 which was released in 2002. Since then users have been encouraged to add `from __future__ import division` to any and all files which use the `/` and `//` operators or to be running the interpreter with the `-Q` flag. If you have not been doing this then you will need to go through your code and do two things:

1. Add `from __future__ import division` to your files
2. Update any division operator as necessary to either use `//` to use floor division or continue using `/` and expect a float

The reason that `/` isn’t simply translated to `//` automatically is that if an object defines its own `__div__` method but not `__floordiv__` then your code would begin to fail.

**Text versus binary data**

In Python 2 you could use the `str` type for both text and binary data. Unfortunately this confluence of two different concepts could lead to brittle code which sometimes worked for either kind of data, sometimes not. It also could lead to confusing APIs if people didn’t explicitly state that something that accepted `str` accepted either text
or binary data instead of one specific type. This complicated the situation especially for anyone supporting multiple languages as APIs wouldn’t bother explicitly supporting `unicode` when they claimed text data support.

To make the distinction between text and binary data clearer and more pronounced, Python 3 did what most languages created in the age of the internet have done and made text and binary data distinct types that cannot blindly be mixed together (Python predates widespread access to the internet). For any code that only deals with text or only binary data, this separation doesn’t pose an issue. But for code that has to deal with both, it does mean you might have to now care about when you are using text compared to binary data, which is why this cannot be entirely automated.

To start, you will need to decide which APIs take text and which take binary (it is highly recommended you don’t design APIs that can take both due to the difficulty of keeping the code working; as stated earlier it is difficult to do well). In Python 2 this means making sure the APIs that take text can work with `unicode` in Python 2 and those that work with binary data work with the `bytes` type from Python 3 and thus a subset of `str` in Python 2 (which the `bytes` type in Python 2 is an alias for). Usually the biggest issue is realizing which methods exist for which types in Python 2 & 3 simultaneously (for text that’s `unicode` in Python 2 and `str` in Python 3, for binary that’s `str`/`bytes` in Python 2 and `bytes` in Python 3). The following table lists the unique methods of each data type across Python 2 & 3 (e.g., the `decode()` method is usable on the equivalent binary data type in either Python 2 or 3, but it can’t be used by the text data type consistently between Python 2 and 3 because `str` in Python 3 doesn’t have the method).

<table>
<thead>
<tr>
<th>Text data</th>
<th>Binary data</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__mod__</code> (% operator)</td>
<td></td>
</tr>
</tbody>
</table>
Making the distinction easier to handle can be accomplished by encoding and decoding between binary data and text at the edge of your code. This means that when you receive text in binary data, you should immediately decode it. And if your code needs to send text as binary data then encode it as late as possible. This allows your code to work with only text internally and thus eliminates having to keep track of what type of data you are working with.

The next issue is making sure you know whether the string literals in your code represent text or binary data. At minimum you should add a `b` prefix to any literal that presents binary data. For text you should either use the `from __future__ import unicode_literals` statement or add a `u` prefix to the text literal.

As part of this dichotomy you also need to be careful about opening files. Unless you have been working on Windows, there is a chance you have not always bothered to add the `b` mode when opening a binary file (e.g., `rb` for binary reading). Under Python 3, binary files and text files are clearly distinct and mutually incompatible; see the `io` module for details. Therefore, you must make a decision of whether a file will be used for binary access (allowing binary data to be read and/or written) or text access (allowing text data to be read and/or written). You should also use `io.open()` for opening files instead of the built-in `open()` function as the `io` module is consistent from Python 2 to 3 while the built-in `open()` function is not (in Python 3 it’s actually `io.open()`).
The constructors of both `str` and `bytes` have different semantics for the same arguments between Python 2 & 3. Passing an integer to `bytes` in Python 2 will give you the string representation of the integer: `bytes(3) == '3'`. But in Python 3, an integer argument to `bytes` will give you a bytes object as long as the integer specified, filled with null bytes: `bytes(3) == b'\x00\x00\x00'`. A similar worry is necessary when passing a bytes object to `str`. In Python 2 you just get the bytes object back: `str(b'3') == b'3'`. But in Python 3 you get the string representation of the bytes object: `str(b'3') == "b'3'"`.

Finally, the indexing of binary data requires careful handling (slicing does not require any special handling). In Python 2, `b'123'[1] == b'2'` while in Python 3 `b'123'[1] == 50`. Because binary data is simply a collection of binary numbers, Python 3 returns the integer value for the byte you index on. But in Python 2 because `bytes == str`, indexing returns a one-item slice of bytes. The six project has a function named `six.indexbytes()` which will return an integer like in Python 3: `six.indexbytes(b'123', 1)`.

To summarize:

1. Decide which of your APIs take text and which take binary data
2. Make sure that your code that works with text also works with `unicode` and code for binary data works with `bytes` in Python 2 (see the table above for what methods you cannot use for each type)
3. Mark all binary literals with a `b` prefix, use a `u` prefix or `__future__` import statement for text literals
4. Decode binary data to text as soon as possible, encode text as binary data as late as possible
5. Open files using `io.open()` and make sure to specify the `b` mode when appropriate
6. Be careful when indexing binary data

Prevent compatibility regressions

Once you have fully translated your code to be compatible with Python 3, you will want to make sure your code doesn't regress and stop working under Python 3. This is especially true if you have a dependency which is blocking you from actually running under Python 3 at the moment.

To help with staying compatible, any new modules you create should have at least the following block of code at the top of it:

```python
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals
```

You can also run Python 2 with the `-3` flag to be warned about various compatibility issues your code triggers during execution. If you turn warnings into errors with `-Werror` then you can make sure that you don’t accidentally miss a warning.

You can also use the Pylint project and its `--py3k` flag to lint your code to receive warnings when your code begins to deviate from Python 3 compatibility. This also prevents you from having to run Modernize or Futurize over your code regularly to catch compatibility regressions. This does require you only support Python 2.7 and Python 3.4 or newer as that is Pylint’s minimum Python version support.

Check which dependencies block your transition

After you have made your code compatible with Python 3 you should begin to care about whether your dependencies have also
been ported. The caniusepython3 project was created to help you determine which projects – directly or indirectly – are blocking you from supporting Python 3. There is both a command-line tool as well as a web interface at https://caniusepython3.com.

The project also provides code which you can integrate into your test suite so that you will have a failing test when you no longer have dependencies blocking you from using Python 3. This allows you to avoid having to manually check your dependencies and to be notified quickly when you can start running on Python 3.

**Update your setup.py file to denote Python 3 compatibility**

Once your code works under Python 3, you should update the classifiers in your `setup.py` to contain `Programming Language :: Python :: 3` and to not specify sole Python 2 support. This will tell anyone using your code that you support Python 2 and 3. Ideally you will also want to add classifiers for each major/minor version of Python you now support.

**Use continuous integration to stay compatible**

Once you are able to fully run under Python 3 you will want to make sure your code always works under both Python 2 & 3. Probably the best tool for running your tests under multiple Python interpreters is `tox`. You can then integrate tox with your continuous integration system so that you never accidentally break Python 2 or 3 support.

You may also want to use the `-bb` flag with the Python 3 interpreter to trigger an exception when you are comparing bytes to strings. Usually it's simply `False`, but if you made a mistake in your separation of text/binary data handling you may be accidentally comparing text and binary data. This flag will raise an exception
when that occurs to help track down such cases.

And that’s mostly it! At this point your code base is compatible with both Python 2 and 3 simultaneously. Your testing will also be set up so that you don’t accidentally break Python 2 or 3 compatibility regardless of which version you typically run your tests under while developing.
Dropping Python 2 support completely

If you are able to fully drop support for Python 2, then the steps required to transition to Python 3 simplify greatly.

1. Update your code to only support Python 2.7
2. Make sure you have good test coverage (*coverage.py* can help)
3. Learn the differences between Python 2 & 3
4. Use *2to3* to rewrite your code to run only under Python 3

After this your code will be fully Python 3 compliant but in a way that is not supported by Python 2. You should also update the classifiers in your *setup.py* to contain `Programming Language :: Python :: 3 :: Only`. 
Porting Extension Modules to Python 3

author: Benjamin Peterson

Abstract

Although changing the C-API was not one of Python 3's objectives, the many Python-level changes made leaving Python 2's API intact impossible. In fact, some changes such as `int()` and `long()` unification are more obvious on the C level. This document endeavors to document incompatibilities and how they can be worked around.
Conditional compilation

The easiest way to compile only some code for Python 3 is to check if `PY_MAJOR_VERSION` is greater than or equal to 3.

```c
#if PY_MAJOR_VERSION >= 3
#define IS_PY3K
#endif
```

API functions that are not present can be aliased to their equivalents within conditional blocks.
Changes to Object APIs

Python 3 merged together some types with similar functions while cleanly separating others.

str/unicode Unification

Python 3’s `str()` type is equivalent to Python 2’s `unicode()`; the C functions are called `PyUnicode_*` for both. The old 8-bit string type has become `bytes()`, with C functions called `PyBytes_*`. Python 2.6 and later provide a compatibility header, `bytesobject.h`, mapping `PyBytes` names to `PyString` ones. For best compatibility with Python 3, `PyUnicode` should be used for textual data and `PyBytes` for binary data. It’s also important to remember that `PyBytes` and `PyUnicode` in Python 3 are not interchangeable like `PyString` and `PyUnicode` are in Python 2. The following example shows best practices with regards to `PyUnicode`, `PyString`, and `PyBytes`.

```c
#include "stdlib.h"
#include "Python.h"
#include "bytesobject.h"

/* text example */
static PyObject *
say_hello(PyObject *self, PyObject *args) {
    PyObject *name, *result;

    if (!PyArg_ParseTuple(args, "U:say_hello", &name))
        return NULL;

    result = PyUnicode_FromFormat("Hello, %S!", name);
    return result;
}

/* just a forward */
static char *
do_encode(PyObject *);
```
/* bytes example */
static PyObject *
encode_object(PyObject *self, PyObject *args) {
    char *encoded;
    PyObject *result, *myobj;

    if (!PyArg_ParseTuple(args, "O:encode_object", &myobj))
        return NULL;

    encoded = do_encode(myobj);
    if (encoded == NULL)
        return NULL;
    result = PyBytes_FromString(encoded);
    free(encoded);
    return result;
}

long/int Unification

Python 3 has only one integer type, \texttt{int()}. But it actually corresponds to Python 2’s \texttt{long()} type—the \texttt{int()} type used in Python 2 was removed. In the C-API, \texttt{PyInt_*} functions are replaced by their \texttt{PyLong_*} equivalents.
Module initialization and state

Python 3 has a revamped extension module initialization system. (See PEP 3121.) Instead of storing module state in globals, they should be stored in an interpreter specific structure. Creating modules that act correctly in both Python 2 and Python 3 is tricky. The following simple example demonstrates how.

```c
#include "Python.h"

struct module_state {
    PyObject *error;
};

#if PY_MAJOR_VERSION >= 3
#define GETSTATE(m) ((struct module_state*)PyModule_GetState(m))
#else
#define GETSTATE(m) (&_state)
#endif

static struct module_state _state;

static PyObject *
error_out(PyObject *m) {
    struct module_state *st = GETSTATE(m);
    PyErr_SetString(st->error, "something bad happened");
    return NULL;
}

static PyMethodDef myextension_methods[] = {
    {"error_out", (PyCFunction)error_out, METH_NOARGS, NULL},
    {NULL, NULL}
};

#if PY_MAJOR_VERSION >= 3
static int myextension_traverse(PyObject *m, visitproc visit, void *v) {
    Py_VISIT(GETSTATE(m)->error);
    return 0;
}
#else
static int myextension_traverse(PyObject *m, visitproc visit, void *v) {
    Py_VISIT(GETSTATE(m)->error);
    return 0;
}
#endif

static int myextension_clear(PyObject *m) {
    Py_CLEAR(GETSTATE(m)->error);
}
```
return 0;
}

static struct PyModuleDef moduledef = {
  PyModuleDef_HEAD_INIT,
  "myextension",
  NULL,
  sizeof(struct module_state),
  myextension_methods,
  NULL,
  myextension_traverse,
  myextension_clear,
  NULL
};

#define INITERROR return NULL

PyMODINIT_FUNC
PyInit_myextension(void)
{
  #if PY_MAJOR_VERSION >= 3
    PyObject *module = PyModule_Create(&moduledef);
  #else
    PyObject *module = Py_InitModule("myextension",
  #endif

    if (module == NULL)
      INITERROR;

    struct module_state *st = GETSTATE(module);

    st->error = PyErr_NewException("myextension.Error", NULL, NULL);
    if (st->error == NULL) {
      Py_DECREF(module);
      INITERROR;
    }

  #if PY_MAJOR_VERSION >= 3
    return module;
  #endif
}

#include "readline/readline.h"
The `capsule` object was introduced in Python 3.1 and 2.7 to replace `CObject`. CObjects were useful, but the `CObject` API was problematic: it didn't permit distinguishing between valid CObjects, which allowed mismatched CObjects to crash the interpreter, and some of its APIs relied on undefined behavior in C. (For further reading on the rationale behind Capsules, please see issue 5630.)

If you're currently using CObjects, and you want to migrate to 3.1 or newer, you'll need to switch to Capsules. `CObject` was deprecated in 3.1 and 2.7 and completely removed in Python 3.2. If you only support 2.7, or 3.1 and above, you can simply switch to `Capsule`. If you need to support Python 3.0, or versions of Python earlier than 2.7, you'll have to support both CObjects and Capsules. (Note that Python 3.0 is no longer supported, and it is not recommended for production use.)

The following example header file `capsulethunk.h` may solve the problem for you. Simply write your code against the `Capsule` API and include this header file after `Python.h`. Your code will automatically use Capsules in versions of Python with Capsules, and switch to CObjects when Capsules are unavailable.

`capsulethunk.h` simulates Capsules using CObjects. However, `CObject` provides no place to store the capsule's “name”. As a result the simulated `Capsule` objects created by `capsulethunk.h` behave slightly differently from real Capsules. Specifically:

- The name parameter passed in to `PyCapsule_New()` is ignored.
- The name parameter passed in to `PyCapsule_IsValid()` and
**PyCapsule_GetPointer()** is ignored, and no error checking of the name is performed.

- **PyCapsule_GetName()** always returns NULL.
- **PyCapsule_SetName()** always raises an exception and returns failure. (Since there's no way to store a name in a CObject, noisy failure of **PyCapsule_SetName()** was deemed preferable to silent failure here. If this is inconvenient, feel free to modify your local copy as you see fit.)

You can find **capsulethunk.h** in the Python source distribution as **Doc/includes/capsulethunk.h**. We also include it here for your convenience:

```c
#ifndef __CAPSULETHUNK_H
#define __CAPSULETHUNK_H

#if (  (PY_VERSION_HEX < 0x02070000)   
  ||  ((PY_VERSION_HEX >= 0x03000000)   
        && (PY_VERSION_HEX < 0x03010000)) )

#define __PyCapsule_GetField(capsule, field, default_value)  
  (PyCapsule_CheckExact(capsule)  
        ? (((PyCObject *)capsule)->field)  
          : (default_value)  
  )

#define __PyCapsule_SetField(capsule, field, value)  
  (PyCapsule_CheckExact(capsule)  
        ? (((PyCObject *)capsule)->field = value), 1  
          : 0  
  )

#define PyCapsule_Type PyCObject_Type

#define PyCapsule_CheckExact(capsule) (PyCObject_Check(capsule)
#define PyCapsule_IsValid(capsule, name) (PyCObject_Check(capsule)

#define PyCapsule_New(pointer, name, destructor)  
  (PyCObject_FromVoidPtr(pointer, destructor))
```

#define PyCapsule_GetPointer(capsule, name) \  (PyObject_AsVoidPtr(capsule))

/* Don't call PyCObject_SetPointer here, it fails if there's a */
#define PyCapsule_SetPointer(capsule, pointer) \  __PyCapsule_SetField(capsule, cobject, pointer)

#define PyCapsule_GetDestructor(capsule) \  __PyCapsule_GetField(capsule, destructor)
#define PyCapsule_SetDestructor(capsule, dtor) \  __PyCapsule_SetField(capsule, destructor, dtor)

/*
 * Sorry, there's simply no place
 * to store a Capsule "name" in a CObject.
 */
#define PyCapsuleGetName(capsule) NULL

static int
PyCapsuleSetName(PyObject *capsule, const char *unused)
{
    unused = unused;
    PyErr_SetString(PyExc_NotImplementedError,
        "can't use PyCapsule_SetName with CObjects");
    return 1;
}

#define PyCapsuleGetContext(capsule) \  __PyCapsuleGetField(capsule, descr)
#define PyCapsuleSetContext(capsule, context) \  __PyCapsuleSetField(capsule, descr, context)

static void *
PyCapsuleImport(const char *name, int no_block)
{
    PyObject *object = NULL;
    void *return_value = NULL;
    char *trace;
    size_t name_length = (strlen(name) + 1) * sizeof(char);
char *name_dup = (char *)PyMem_MALLOC(name_length);

if (!name_dup) {
    return NULL;
}

memcpy(name_dup, name, name_length);

trace = name_dup;
while (trace) {
    char *dot = strchr(trace, '.');
    if (dot) {
        *dot++ = '\0';
    }

    if (object == NULL) {
        if (no_block) {
            object = PyImport.ImportModuleNoBlock(trace);
        } else {
            object = PyImport.ImportModule(trace);
            if (!object) {
                PyErr_Format(PyExc_ImportError,
                    "PyCapsule_Import could not "
                    "import module \"%s\"", trace);
            }
        }
    } else {
        PyObject *object2 = PyObject_GetAttrString(object,
            Py_DECREF(object);
        object = object2;
    }
    if (!object) {
        goto EXIT;
    }

    trace = dot;
}

if (PyObject_Check(object)) {
    PyObject *object = (PyObject *)object;
    return_value = object->cobject;
} else {
    PyErr_Format(PyExc_AttributeError,
        "PyCapsule_Import \"%s\" is not valid", name);
}
EXIT:
    Py_XDECREF(object);
    if (name_dup) {
        PyMem_FREE(name_dup);
    }
    return return_value;
}
#endif /* #if PY_VERSION_HEX < 0x02070000 */
#endif /* __CAPSULETHUNK_H */
Other options

If you are writing a new extension module, you might consider Cython. It translates a Python-like language to C. The extension modules it creates are compatible with Python 3 and Python 2.
Abstract

This document describes how to write text-mode programs with Python 2.x, using the `curses` extension module to control the display.
What is curses?

The curses library supplies a terminal-independent screen-painting and keyboard-handling facility for text-based terminals; such terminals include VT100s, the Linux console, and the simulated terminal provided by X11 programs such as xterm and rxvt. Display terminals support various control codes to perform common operations such as moving the cursor, scrolling the screen, and erasing areas. Different terminals use widely differing codes, and often have their own minor quirks.

In a world of X displays, one might ask “why bother”? It's true that character-cell display terminals are an obsolete technology, but there are niches in which being able to do fancy things with them are still valuable. One is on small-footprint or embedded Unixes that don’t carry an X server. Another is for tools like OS installers and kernel configurators that may have to run before X is available.

The curses library hides all the details of different terminals, and provides the programmer with an abstraction of a display, containing multiple non-overlapping windows. The contents of a window can be changed in various ways—adding text, erasing it, changing its appearance—and the curses library will automagically figure out what control codes need to be sent to the terminal to produce the right output.

The curses library was originally written for BSD Unix; the later System V versions of Unix from AT&T added many enhancements and new functions. BSD curses is no longer maintained, having been replaced by ncurses, which is an open-source implementation of the AT&T interface. If you're using an open-source Unix such as Linux or FreeBSD, your system almost certainly uses ncurses. Since most current commercial Unix versions are based on System V code, all the functions described here will probably be available. The older
versions of curses carried by some proprietary Unixes may not support everything, though.

No one has made a Windows port of the curses module. On a Windows platform, try the Console module written by Fredrik Lundh. The Console module provides cursor-addressable text output, plus full support for mouse and keyboard input, and is available from http://effbot.org/zone/console-index.htm.

The Python curses module

Thy Python module is a fairly simple wrapper over the C functions provided by curses; if you're already familiar with curses programming in C, it's really easy to transfer that knowledge to Python. The biggest difference is that the Python interface makes things simpler, by merging different C functions such as `addstr()`, `mvaddstr()`, `mvwaddstr()`, into a single `addstr()` method. You'll see this covered in more detail later.

This HOWTO is simply an introduction to writing text-mode programs with curses and Python. It doesn't attempt to be a complete guide to the curses API; for that, see the Python library guide’s section on ncurses, and the C manual pages for ncurses. It will, however, give you the basic ideas.
Starting and ending a curses application

Before doing anything, curses must be initialized. This is done by calling the `initscr()` function, which will determine the terminal type, send any required setup codes to the terminal, and create various internal data structures. If successful, `initscr()` returns a window object representing the entire screen; this is usually called `stdscr`, after the name of the corresponding C variable.

```python
import curses
stdscr = curses.initscr()
```

Usually curses applications turn off automatic echoing of keys to the screen, in order to be able to read keys and only display them under certain circumstances. This requires calling the `noecho()` function.

```python
curses.noecho()
```

Applications will also commonly need to react to keys instantly, without requiring the Enter key to be pressed; this is called `cbreak` mode, as opposed to the usual buffered input mode.

```python
curses.cbreak()
```

Terminals usually return special keys, such as the cursor keys or navigation keys such as Page Up and Home, as a multibyte escape sequence. While you could write your application to expect such sequences and process them accordingly, curses can do it for you, returning a special value such as `curses.KEY_LEFT`. To get curses to do the job, you’ll have to enable keypad mode.

```python
stdscr.keypad(1)
```

Terminating a curses application is much easier than starting one.
You'll need to call

```python
curses.nocbreak(); stdscr.keypad(0); curses.echo()
```

to reverse the curses-friendly terminal settings. Then call the `endwin()` function to restore the terminal to its original operating mode.

```python
curses.endwin()
```

A common problem when debugging a curses application is to get your terminal messed up when the application dies without restoring the terminal to its previous state. In Python this commonly happens when your code is buggy and raises an uncaught exception. Keys are no longer echoed to the screen when you type them, for example, which makes using the shell difficult.

In Python you can avoid these complications and make debugging much easier by importing the `curses.wrapper()` function. It takes a callable and does the initializations described above, also initializing colors if color support is present. It then runs your provided callable and finally deinitializes appropriately. The callable is called inside a try-catch clause which catches exceptions, performs curses deinitialization, and then passes the exception upwards. Thus, your terminal won’t be left in a funny state on exception.
Windows and Pads

Windows are the basic abstraction in curses. A window object represents a rectangular area of the screen, and supports various methods to display text, erase it, allow the user to input strings, and so forth.

The `stdscr` object returned by the `initscr()` function is a window object that covers the entire screen. Many programs may need only this single window, but you might wish to divide the screen into smaller windows, in order to redraw or clear them separately. The `newwin()` function creates a new window of a given size, returning the new window object.

```c
begin_x = 20; begin_y = 7
height = 5; width = 40
win = curses.newwin(height, width, begin_y, begin_x)
```

A word about the coordinate system used in curses: coordinates are always passed in the order y,x, and the top-left corner of a window is coordinate (0,0). This breaks a common convention for handling coordinates, where the x coordinate usually comes first. This is an unfortunate difference from most other computer applications, but it's been part of curses since it was first written, and it's too late to change things now.

When you call a method to display or erase text, the effect doesn't immediately show up on the display. This is because curses was originally written with slow 300-baud terminal connections in mind; with these terminals, minimizing the time required to redraw the screen is very important. This lets curses accumulate changes to the screen, and display them in the most efficient manner. For example, if your program displays some characters in a window, and then clears the window, there's no need to send the original characters
because they'd never be visible.

Accordingly, curses requires that you explicitly tell it to redraw windows, using the `refresh()` method of window objects. In practice, this doesn't really complicate programming with curses much. Most programs go into a flurry of activity, and then pause waiting for a keypress or some other action on the part of the user. All you have to do is to be sure that the screen has been redrawn before pausing to wait for user input, by simply calling `stdscr.refresh()` or the `refresh()` method of some other relevant window.

A pad is a special case of a window; it can be larger than the actual display screen, and only a portion of it displayed at a time. Creating a pad simply requires the pad's height and width, while refreshing a pad requires giving the coordinates of the on-screen area where a subsection of the pad will be displayed.

```python
pad = curses.newpad(100, 100)
# These loops fill the pad with letters; this is
# explained in the next section
for y in range(0, 100):
    for x in range(0, 100):
        try:
            pad.addch(y,x, ord('a') + (x*x+y*y) % 26)
        except curses.error:
            pass

# Displays a section of the pad in the middle of the screen
pad.refresh(0, 0, 5, 5, 20, 75)
```

The `refresh()` call displays a section of the pad in the rectangle extending from coordinate (5,5) to coordinate (20,75) on the screen; the upper left corner of the displayed section is coordinate (0,0) on the pad. Beyond that difference, pads are exactly like ordinary windows and support the same methods.

If you have multiple windows and pads on screen there is a more efficient way to go, which will prevent annoying screen flicker at
refresh time. Use the `noutrefresh()` method of each window to update the data structure representing the desired state of the screen; then change the physical screen to match the desired state in one go with the function `doupdate()`. The normal `refresh()` method calls `doupdate()` as its last act.
Displaying Text

From a C programmer’s point of view, curses may sometimes look like a twisty maze of functions, all subtly different. For example, addstr() displays a string at the current cursor location in the stdscr window, while mvaddstr() moves to a given y,x coordinate first before displaying the string. waddstr() is just like addstr(), but allows specifying a window to use, instead of using stdscr by default. mvwaddstr() follows similarly.

Fortunately the Python interface hides all these details; stdscr is a window object like any other, and methods like addstr() accept multiple argument forms. Usually there are four different forms.

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str or ch</td>
<td>Display the string str or character ch at the current position</td>
</tr>
<tr>
<td>str or ch, attr</td>
<td>Display the string str or character ch, using attribute attr at the current position</td>
</tr>
<tr>
<td>y, x, str or ch</td>
<td>Move to position y,x within the window, and display str or ch</td>
</tr>
<tr>
<td>y, x, str or ch, attr</td>
<td>Move to position y,x within the window, and display str or ch, using attribute attr</td>
</tr>
</tbody>
</table>

Attributes allow displaying text in highlighted forms, such as in boldface, underline, reverse code, or in color. They’ll be explained in more detail in the next subsection.

The addstr() function takes a Python string as the value to be displayed, while the addch() functions take a character, which can be either a Python string of length 1 or an integer. If it's a string, you’re
limited to displaying characters between 0 and 255. SVr4 curses provides constants for extension characters; these constants are integers greater than 255. For example, `ACS_PLMINUS` is a +/- symbol, and `ACS_ULCORNER` is the upper left corner of a box (handy for drawing borders).

Windows remember where the cursor was left after the last operation, so if you leave out the y,x coordinates, the string or character will be displayed wherever the last operation left off. You can also move the cursor with the `move(y, x)` method. Because some terminals always display a flashing cursor, you may want to ensure that the cursor is positioned in some location where it won’t be distracting; it can be confusing to have the cursor blinking at some apparently random location.

If your application doesn’t need a blinking cursor at all, you can call `curs_set(0)` to make it invisible. Equivalently, and for compatibility with older curses versions, there’s a `leaveok(bool)` function. When `bool` is true, the curses library will attempt to suppress the flashing cursor, and you won’t need to worry about leaving it in odd locations.

## Attributes and Color

Characters can be displayed in different ways. Status lines in a text-based application are commonly shown in reverse video; a text viewer may need to highlight certain words. curses supports this by allowing you to specify an attribute for each cell on the screen.

An attribute is an integer, each bit representing a different attribute. You can try to display text with multiple attribute bits set, but curses doesn’t guarantee that all the possible combinations are available, or that they’re all visually distinct. That depends on the ability of the terminal being used, so it’s safest to stick to the most commonly available attributes, listed here.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_BLINK</td>
<td>Blinking text</td>
</tr>
<tr>
<td>A_BOLD</td>
<td>Extra bright or bold text</td>
</tr>
<tr>
<td>A_DIM</td>
<td>Half bright text</td>
</tr>
<tr>
<td>A_REVERSE</td>
<td>Reverse-video text</td>
</tr>
<tr>
<td>A_STANDOUT</td>
<td>The best highlighting mode available</td>
</tr>
<tr>
<td>A_UNDERLINE</td>
<td>Underlined text</td>
</tr>
</tbody>
</table>

So, to display a reverse-video status line on the top line of the screen, you could code:

```python
stdscr.addstr(0, 0, "Current mode: Typing mode", curses.A_REVERSE)
stdscr.refresh()
```

The curses library also supports color on those terminals that provide it. The most common such terminal is probably the Linux console, followed by color xterms.

To use color, you must call the `start_color()` function soon after calling `initscr()`, to initialize the default color set (the `curses.wrapper.wrapper()` function does this automatically). Once that's done, the `has_colors()` function returns TRUE if the terminal in use can actually display color. (Note: curses uses the American spelling ‘color’, instead of the Canadian/British spelling ‘colour’. If you’re used to the British spelling, you’ll have to resign yourself to misspelling it for the sake of these functions.)

The curses library maintains a finite number of color pairs, containing a foreground (or text) color and a background color. You can get the attribute value corresponding to a color pair with the `color_pair()` function; this can be bitwise-OR’ed with other attributes such as `A_REVERSE`, but again, such combinations are not guaranteed to work.
on all terminals.

An example, which displays a line of text using color pair 1:

```c
stdscr.addstr("Pretty text", curses.color_pair(1))
stdscr.refresh()
```

As I said before, a color pair consists of a foreground and background color. `start_color()` initializes 8 basic colors when it activates color mode. They are: 0:black, 1:red, 2:green, 3:yellow, 4:blue, 5:magenta, 6:cyan, and 7:white. The curses module defines named constants for each of these colors: `curses.COLOR_BLACK`, `curses.COLOR_RED`, and so forth.

The `init_pair(n, f, b)` function changes the definition of color pair `n`, to foreground color `f` and background color `b`. Color pair 0 is hard-wired to white on black, and cannot be changed.

Let's put all this together. To change color 1 to red text on a white background, you would call:

```c
 curses.init_pair(1, curses.COLOR_RED, curses.COLOR_WHITE)
```

When you change a color pair, any text already displayed using that color pair will change to the new colors. You can also display new text in this color with:

```c
stdscr.addstr(0,0, "RED ALERT!", curses.color_pair(1))
```

Very fancy terminals can change the definitions of the actual colors to a given RGB value. This lets you change color 1, which is usually red, to purple or blue or any other color you like. Unfortunately, the Linux console doesn’t support this, so I’m unable to try it out, and can’t provide any examples. You can check if your terminal can do this by calling `can_change_color()`, which returns TRUE if the
capability is there. If you're lucky enough to have such a talented terminal, consult your system's man pages for more information.
The curses library itself offers only very simple input mechanisms. Python’s support adds a text-input widget that makes up some of the lack.

The most common way to get input to a window is to use its `getch()` method. `getch()` pauses and waits for the user to hit a key, displaying it if `echo()` has been called earlier. You can optionally specify a coordinate to which the cursor should be moved before pausing.

It’s possible to change this behavior with the method `nodelay()`. After `nodelay(1), getch()` for the window becomes non-blocking and returns `curses.ERR` (a value of -1) when no input is ready. There’s also a `halfdelay()` function, which can be used to (in effect) set a timer on each `getch();` if no input becomes available within a specified delay (measured in tenths of a second), curses raises an exception.

The `getch()` method returns an integer; if it’s between 0 and 255, it represents the ASCII code of the key pressed. Values greater than 255 are special keys such as Page Up, Home, or the cursor keys. You can compare the value returned to constants such as `curses.KEY_PPAge, curses.KEY_HOME,` or `curses.KEY_LEFT`. Usually the main loop of your program will look something like this:

```python
while 1:
    c = stdscr.getch()
    if c == ord('p'):
        PrintDocument()
    elif c == ord('q'):
        break  # Exit the while()
    elif c == curses.KEY_HOME:
        x = y = 0
```
The `curses.ascii` module supplies ASCII class membership functions that take either integer or 1-character-string arguments; these may be useful in writing more readable tests for your command interpreters. It also supplies conversion functions that take either integer or 1-character-string arguments and return the same type. For example, `curses.ascii.ctrl()` returns the control character corresponding to its argument.

There's also a method to retrieve an entire string, `getstr()`. It isn't used very often, because its functionality is quite limited; the only editing keys available are the backspace key and the Enter key, which terminates the string. It can optionally be limited to a fixed number of characters.

```python
curses.echo()  # Enable echoing of characters

# Get a 15-character string, with the cursor on the top line
s = stdscr.getstr(0, 0, 15)
```

The Python `curses.textpad` module supplies something better. With it, you can turn a window into a text box that supports an Emacs-like set of keybindings. Various methods of `Textbox` class support editing with input validation and gathering the edit results either with or without trailing spaces. See the library documentation on `curses.textpad` for the details.
This HOWTO didn’t cover some advanced topics, such as screen-scraping or capturing mouse events from an xterm instance. But the Python library page for the curses modules is now pretty complete. You should browse it next.

If you’re in doubt about the detailed behavior of any of the ncurses entry points, consult the manual pages for your curses implementation, whether it's ncurses or a proprietary Unix vendor’s. The manual pages will document any quirks, and provide complete lists of all the functions, attributes, and \texttt{acs\_} characters available to you.

Because the curses API is so large, some functions aren't supported in the Python interface, not because they’re difficult to implement, but because no one has needed them yet. Feel free to add them and then submit a patch. Also, we don't yet have support for the menu library associated with ncurses; feel free to add that.

If you write an interesting little program, feel free to contribute it as another demo. We can always use more of them!

The ncurses FAQ: \url{http://invisible-island.net/ncurses/ncurses.faq.html}
Descriptor HowTo Guide

Author: Raymond Hettinger
Contact: <python at rcn dot com>

Contents

- Descriptor HowTo Guide
  - Abstract
  - Definition and Introduction
  - Descriptor Protocol
  - Invoking Descriptors
  - Descriptor Example
  - Properties
  - Functions and Methods
  - Static Methods and Class Methods
Abstract

Defines descriptors, summarizes the protocol, and shows how descriptors are called. Examines a custom descriptor and several built-in python descriptors including functions, properties, static methods, and class methods. Shows how each works by giving a pure Python equivalent and a sample application.

Learning about descriptors not only provides access to a larger toolset, it creates a deeper understanding of how Python works and an appreciation for the elegance of its design.
Definition and Introduction

In general, a descriptor is an object attribute with “binding behavior”, one whose attribute access has been overridden by methods in the descriptor protocol. Those methods are `__get__()`, `__set__()`, and `__delete__()`. If any of those methods are defined for an object, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object’s dictionary. For instance, `a.x` has a lookup chain starting with `a.__dict__['x']`, then `type(a).__dict__['x']`, and continuing through the base classes of `type(a)` excluding metaclasses. If the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined. Note that descriptors are only invoked for new style objects or classes (a class is new style if it inherits from `object` or `type`).

Descriptors are a powerful, general purpose protocol. They are the mechanism behind properties, methods, static methods, class methods, and `super()`. They are used throughout Python itself to implement the new style classes introduced in version 2.2. Descriptors simplify the underlying C-code and offer a flexible set of new tools for everyday Python programs.
Descriptor Protocol

```python
descr.__get__(self, obj, type=None) --> value

descr.__set__(self, obj, value) --> None

descr.__delete__(self, obj) --> None
```

That is all there is to it. Define any of these methods and an object is considered a descriptor and can override default behavior upon being looked up as an attribute.

If an object defines both `__get__()` and `__set__()`, it is considered a data descriptor. Descriptors that only define `__get__()` are called non-data descriptors (they are typically used for methods but other uses are possible).

Data and non-data descriptors differ in how overrides are calculated with respect to entries in an instance's dictionary. If an instance's dictionary has an entry with the same name as a data descriptor, the data descriptor takes precedence. If an instance's dictionary has an entry with the same name as a non-data descriptor, the dictionary entry takes precedence.

To make a read-only data descriptor, define both `__get__()` and `__set__()` with the `__set__()` raising an `AttributeError` when called. Defining the `__set__()` method with an exception raising placeholder is enough to make it a data descriptor.
Invoking Descriptors

A descriptor can be called directly by its method name. For example, `d.__get__(obj)`.

Alternatively, it is more common for a descriptor to be invoked automatically upon attribute access. For example, `obj.d` looks up `d` in the dictionary of `obj`. If `d` defines the method `__get__()`, then `d.__get__(obj)` is invoked according to the precedence rules listed below.

The details of invocation depend on whether `obj` is an object or a class. Either way, descriptors only work for new style objects and classes. A class is new style if it is a subclass of `object`.

For objects, the machinery is in `object.__getattribute__()` which transforms `b.x` into `type(b).__dict__['x'].__get__(b, type(b))`. The implementation works through a precedence chain that gives data descriptors priority over instance variables, instance variables priority over non-data descriptors, and assigns lowest priority to `__getattr__()` if provided. The full C implementation can be found in `PyObject_GenericGetAttr()` in `Objects/object.c`.

For classes, the machinery is in `type.__getattribute__()` which transforms `B.x` into `B.__dict__['x'].__get__(None, B)`. In pure Python, it looks like:

```python
def __getattribute__(self, key):
    "Emulate type_getattro() in Objects/typeobject.c"
    v = object.__getattribute__(self, key)
    if hasattr(v, '__get__'):
        return v.__get__(None, self)
    return v
```
The important points to remember are:

- descriptors are invoked by the `__getattribute__()` method
- overriding `__getattribute__()` prevents automatic descriptor calls
- `__getattribute__()` is only available with new style classes and objects
- `object.__getattribute__()` and `type.__getattribute__()` make different calls to `__get__()`. 
- data descriptors always override instance dictionaries.
- non-data descriptors may be overridden by instance dictionaries.

The object returned by `super()` also has a custom `__getattribute__()` method for invoking descriptors. The call `super(B, obj).m()` searches `obj.__class__.__mro__` for the base class `A` immediately following `B` and then returns `A.__dict__['m'].__get__(obj, B)`. If not a descriptor, `m` is returned unchanged. If not in the dictionary, `m` reverts to a search using `object.__getattribute__()`.

Note, in Python 2.2, `super(B, obj).m()` would only invoke `__get__()` if `m` was a data descriptor. In Python 2.3, non-data descriptors also get invoked unless an old-style class is involved. The implementation details are in `super_getattro()` in `Objects/typeobject.c`.

The details above show that the mechanism for descriptors is embedded in the `__getattribute__()` methods for `object`, `type`, and `super()`. Classes inherit this machinery when they derive from `object` or if they have a meta-class providing similar functionality. Likewise, classes can turn-off descriptor invocation by overriding `__getattribute__()`.
### Descriptor Example

The following code creates a class whose objects are data descriptors which print a message for each get or set. Overriding `__getattribute__()` is alternate approach that could do this for every attribute. However, this descriptor is useful for monitoring just a few chosen attributes:

```python
class RevealAccess(object):
    """A data descriptor that sets and returns values normally and prints a message logging their access. """
    def __init__(self, initval=None, name='var'):
        self.val = initval
        self.name = name
    def __get__(self, obj, objtype):
        print 'Retrieving', self.name
        return self.val
    def __set__(self, obj, val):
        print 'Updating', self.name
        self.val = val
```

```python
>>> class MyClass(object):
...     x = RevealAccess(10, 'var "x"')
...     y = 5
... >>> m = MyClass()
... >>> m.x
Retrieving var "x"
10
>>> m.x = 20
Updating var "x"
>>> m.x
Retrieving var "x"
20
>>> m.y
5
```
The protocol is simple and offers exciting possibilities. Several use cases are so common that they have been packaged into individual function calls. Properties, bound and unbound methods, static methods, and class methods are all based on the descriptor protocol.
Properties

Calling `property()` is a succinct way of building a data descriptor that triggers function calls upon access to an attribute. Its signature is:

```python
property(fget=None, fset=None, fdel=None, doc=None) -> property
```

The documentation shows a typical use to define a managed attribute `x`:

```python
class C(object):
    def getx(self): return self.__x
    def setx(self, value): self.__x = value
    def delx(self): del self.__x
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

To see how `property()` is implemented in terms of the descriptor protocol, here is a pure Python equivalent:

```python
class Property(object):
    "Emulate PyProperty_Type() in Objects/descrobject.c"

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
The *property()* builtin helps whenever a user interface has granted attribute access and then subsequent changes require the intervention of a method.

For instance, a spreadsheet class may grant access to a cell value through `Cell('b10').value`. Subsequent improvements to the program require the cell to be recalculated on every access; however, the programmer does not want to affect existing client code accessing the attribute directly. The solution is to wrap access to the value attribute in a property data descriptor:

```python
class Cell(object):
    
    def _fget__(self, fget):
        return type(self)(fget, self.fset, self.fdel, self.__doc__)
    
    def _fset__(self, obj, value):
        self.fset(obj, value)
        
    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(obj)
        
    def getter(self, fget):
        return type(self)(fget, self.fset, self.fdel, self.__doc__)
    
    def setter(self, fset):
        return type(self)(self.fget, fset, self.fdel, self.__doc__)
    
    def deleter(self, fdel):
        return type(self)(self.fget, self.fset, fdel, self.__doc__)
```
Functions and Methods

Python’s object oriented features are built upon a function based environment. Using non-data descriptors, the two are merged seamlessly.

Class dictionaries store methods as functions. In a class definition, methods are written using `def` and `lambda`, the usual tools for creating functions. The only difference from regular functions is that the first argument is reserved for the object instance. By Python convention, the instance reference is called `self` but may be called `this` or any other variable name.

To support method calls, functions include the `__get__()` method for binding methods during attribute access. This means that all functions are non-data descriptors which return bound or unbound methods depending whether they are invoked from an object or a class. In pure python, it works like this:

```python
class Function(object):
    ...
    def __get__(self, obj, objtype=None):
        "Simulate func_descr_get() in Objects/funcobject.c"
        return types.MethodType(self, obj, objtype)
```

Running the interpreter shows how the function descriptor works in practice:

```python
>>> class D(object):
...    def f(self, x):
...        return x
...    ...
>>> d = D()
>>> D.__dict__['f']    # Stored internally as a function
<function f at 0x00C45070>
>>> D.f                # Get from a class becomes an unbound meth
<unbound method D.f>
```
The output suggests that bound and unbound methods are two different types. While they could have been implemented that way, the actual C implementation of `PyMethod_Type` in `Objects/classobject.c` is a single object with two different representations depending on whether the `im_self` field is set or is `NULL` (the C equivalent of `None`).

Likewise, the effects of calling a method object depend on the `im_self` field. If set (meaning bound), the original function (stored in the `im_func` field) is called as expected with the first argument set to the instance. If unbound, all of the arguments are passed unchanged to the original function. The actual C implementation of `instancemethod_call()` is only slightly more complex in that it includes some type checking.
Non-data descriptors provide a simple mechanism for variations on the usual patterns of binding functions into methods.

To recap, functions have a `__get__()` method so that they can be converted to a method when accessed as attributes. The non-data descriptor transforms an `obj.f(*args)` call into `f(obj, *args)`. Calling `klass.f(*args)` becomes `f(*args)`.

This chart summarizes the binding and its two most useful variants:

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Called from an Object</th>
<th>Called from a Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td><code>f(obj, *args)</code></td>
<td><code>f(*args)</code></td>
</tr>
<tr>
<td>staticmethod</td>
<td><code>f(*args)</code></td>
<td><code>f(*args)</code></td>
</tr>
<tr>
<td>classmethod</td>
<td><code>f(type(obj), *args)</code></td>
<td><code>f(klass, *args)</code></td>
</tr>
</tbody>
</table>

Static methods return the underlying function without changes. Calling either `c.f` or `C.f` is the equivalent of a direct lookup into `object.__getattribute__(c, "f")` or `object.__getattribute__(C, "f")`. As a result, the function becomes identically accessible from either an object or a class.

Good candidates for static methods are methods that do not reference the `self` variable.

For instance, a statistics package may include a container class for experimental data. The class provides normal methods for computing the average, mean, median, and other descriptive statistics that depend on the data. However, there may be useful functions which are conceptually related but do not depend on the data. For instance, `erf(x)` is handy conversion routine that comes
up in statistical work but does not directly depend on a particular dataset. It can be called either from an object or the class: 

\[ \text{s.erf}(1.5) \rightarrow .9332 \text{ or } \text{Sample.erf}(1.5) \rightarrow .9332. \]

Since staticmethods return the underlying function with no changes, the example calls are unexciting:

```python
>>> class E(object):
...     def f(x):
...         print x
...     f = staticmethod(f)
...
>>> print E.f(3)
3
>>> print E().f(3)
3
```

Using the non-data descriptor protocol, a pure Python version of `staticmethod()` would look like this:

```python
class StaticMethod(object):
    "Emulate PyStaticMethod_Type() in Objects/funcobject.c"

    def __init__(self, f):
        self.f = f

    def __get__(self, obj, objtype=None):
        return self.f
```

Unlike static methods, class methods prepend the class reference to the argument list before calling the function. This format is the same for whether the caller is an object or a class:

```python
>>> class E(object):
...     def f(klass, x):
...         return klass.__name__, x
...     f = classmethod(f)
...
>>> print E.f(3)
('E', 3)
>>> print E().f(3)
```
This behavior is useful whenever the function only needs to have a class reference and does not care about any underlying data. One use for classmethods is to create alternate class constructors. In Python 2.3, the classmethod `dict.fromkeys()` creates a new dictionary from a list of keys. The pure Python equivalent is:

```python
class Dict(object):
    ...
    def fromkeys(klass, iterable, value=None):
        """Emulate dict_fromkeys() in Objects/dictobject.c"
        d = klass()
        for key in iterable:
            d[key] = value
        return d
fromkeys = classmethod(fromkeys)
```

Now a new dictionary of unique keys can be constructed like this:

```python
>>> Dict.fromkeys('abracadabra')
{'a': None, 'r': None, 'b': None, 'c': None, 'd': None}
```

Using the non-data descriptor protocol, a pure Python version of `classmethod()` would look like this:

```python
class ClassMethod(object):
    """Emulate PyClassMethod_Type() in Objects/funcobject.c"
    
    def __init__(self, f):
        self.f = f

    def __get__(self, obj, klass=None):
        if klass is None:
            klass = type(obj)
        def newfunc(*args):
            return self.f(klass, *args)
        return newfunc
```
Idioms and Anti-Idioms in Python

Author: Moshe Zadka

This document is placed in the public domain.

Abstract

This document can be considered a companion to the tutorial. It shows how to use Python, and even more importantly, how not to use Python.
## Language Constructs You Should Not Use

While Python has relatively few gotchas compared to other languages, it still has some constructs which are only useful in corner cases, or are plain dangerous.

### from module import *

#### Inside Function Definitions

*from module import * is invalid* inside function definitions. While many versions of Python do not check for the invalidity, it does not make it more valid, no more than having a smart lawyer makes a man innocent. Do not use it like that ever. Even in versions where it was accepted, it made the function execution slower, because the compiler could not be certain which names were local and which were global. In Python 2.1 this construct causes warnings, and sometimes even errors.

#### At Module Level

While it is valid to use *from module import * at module level* it is usually a bad idea. For one, this loses an important property Python otherwise has — you can know where each toplevel name is defined by a simple “search” function in your favourite editor. You also open yourself to trouble in the future, if some module grows additional functions or classes.

One of the most awful questions asked on the newsgroup is why this code:

```python
f = open("www")
f.read()
```
does not work. Of course, it works just fine (assuming you have a file called “www”.) But it does not work if somewhere in the module, the statement from os import * is present. The os module has a function called open() which returns an integer. While it is very useful, shadowing a builtin is one of its least useful properties.

Remember, you can never know for sure what names a module exports, so either take what you need — from module import name1, name2, or keep them in the module and access on a per-need basis — import module; print module.name.

When It Is Just Fine

There are situations in which from module import * is just fine:

- The interactive prompt. For example, from math import * makes Python an amazing scientific calculator.
- When extending a module in C with a module in Python.
- When the module advertises itself as from import * safe.

Unadorned exec, execfile() and friends

The word “unadorned” refers to the use without an explicit dictionary, in which case those constructs evaluate code in the current environment. This is dangerous for the same reasons from import * is dangerous — it might step over variables you are counting on and mess up things for the rest of your code. Simply do not do that.

Bad examples:

```python
g>>> for name in sys.argv[1:]:
  ...   exec "%s=1" % name
>>> def func(s, **kw):
  ...   for var, val in kw.items():
  ...     exec "s.%s=val" % var  # invalid!
```
Good examples:

```python
>>> d = {}
>>> for name in sys.argv[1:]:
...     d[name] = 1
>>> def func(s, **kw):
...     for var, val in kw.items():
...         setattr(s, var, val)
>>> d = {}
>>> execfile("handler.py", d, d)
>>> handle = d['handle']
>>> handle()
```

```python
from module import name1, name2
```

This is a “don’t” which is much weaker than the previous “don’t”s but is still something you should not do if you don’t have good reasons to do that. The reason it is usually a bad idea is because you suddenly have an object which lives in two separate namespaces. When the binding in one namespace changes, the binding in the other will not, so there will be a discrepancy between them. This happens when, for example, one module is reloaded, or changes the definition of a function at runtime.

Bad example:

```python
# foo.py
da = 1

# bar.py
from foo import a
if something():
    a = 2 # danger: foo.a != a
```

Good example:
Python has the `except:` clause, which catches all exceptions. Since every error in Python raises an exception, using `except:` can make many programming errors look like runtime problems, which hinders the debugging process.

The following code shows a great example of why this is bad:

```python
try:
    foo = opne("file")  # misspelled "open"
except:
    sys.exit("could not open file!")
```

The second line triggers a `NameError`, which is caught by the `except` clause. The program will exit, and the error message the program prints will make you think the problem is the readability of "file" when in fact the real error has nothing to do with "file".

A better way to write the above is

```python
try:
    foo = opne("file")
except IOError:
    sys.exit("could not open file")
```

When this is run, Python will produce a traceback showing the `NameError`, and it will be immediately apparent what needs to be fixed.
Because `except:` catches *all* exceptions, including `SystemExit`, `KeyboardInterrupt`, and `GeneratorExit` (which is not an error and should not normally be caught by user code), using a bare `except:` is almost never a good idea. In situations where you need to catch all “normal” errors, such as in a framework that runs callbacks, you can catch the base class for all normal exceptions, `Exception`. Unfortunately in Python 2.x it is possible for third-party code to raise exceptions that do not inherit from `Exception`, so in Python 2.x there are some cases where you may have to use a bare `except:` and manually re-raise the exceptions you don’t want to catch.
Exceptions

Exceptions are a useful feature of Python. You should learn to raise them whenever something unexpected occurs, and catch them only where you can do something about them.

The following is a very popular anti-idiom

```python
def get_status(file):
    if not os.path.exists(file):
        print "file not found"
        sys.exit(1)
    return open(file).readline()
```

Consider the case where the file gets deleted between the time the call to `os.path.exists()` is made and the time `open()` is called. In that case the last line will raise an `IOError`. The same thing would happen if `file` exists but has no read permission. Since testing this on a normal machine on existent and non-existent files makes it seem bugless, the test results will seem fine, and the code will get shipped. Later an unhandled `IOError` (or perhaps some other `EnvironmentError`) escapes to the user, who gets to watch the ugly traceback.

Here is a somewhat better way to do it.

```python
def get_status(file):
    try:
        return open(file).readline()
    except EnvironmentError as err:
        print "Unable to open file: {}".format(err)
        sys.exit(1)
```

In this version, *either* the file gets opened and the line is read (so it works even on flaky NFS or SMB connections), or an error message is printed that provides all the available information on why the open
failed, and the application is aborted.

However, even this version of `get_status()` makes too many assumptions — that it will only be used in a short running script, and not, say, in a long running server. Sure, the caller could do something like

```python
try:
    status = get_status(log)
except SystemExit:
    status = None
```

But there is a better way. You should try to use as few `except` clauses in your code as you can — the ones you do use will usually be inside calls which should always succeed, or a catch-all in a main function.

So, an even better version of `get_status()` is probably

```python
def get_status(file):
    return open(file).readline()
```

The caller can deal with the exception if it wants (for example, if it tries several files in a loop), or just let the exception filter upwards to its caller.

But the last version still has a serious problem — due to implementation details in CPython, the file would not be closed when an exception is raised until the exception handler finishes; and, worse, in other implementations (e.g., Jython) it might not be closed at all regardless of whether or not an exception is raised.

The best version of this function uses the `open()` call as a context manager, which will ensure that the file gets closed as soon as the function returns:
def get_status(file):
    with open(file) as fp:
        return fp.readline()
Using the Batteries

Every so often, people seem to be writing stuff in the Python library again, usually poorly. While the occasional module has a poor interface, it is usually much better to use the rich standard library and data types that come with Python than inventing your own.

A useful module very few people know about is `os.path`. It always has the correct path arithmetic for your operating system, and will usually be much better than whatever you come up with yourself.

Compare:

```python
# ugh!
return dir+"/"+file
# better
return os.path.join(dir, file)
```

More useful functions in `os.path`: `basename()`, `dirname()` and `splitext()`.

There are also many useful built-in functions people seem not to be aware of for some reason: `min()` and `max()` can find the minimum/maximum of any sequence with comparable semantics, for example, yet many people write their own `max()/min()`. Another highly useful function is `reduce()` which can be used to repeatedly apply a binary operation to a sequence, reducing it to a single value. For example, compute a factorial with a series of multiply operations:

```python
>>> n = 4
>>> import operator
>>> reduce(operator.mul, range(1, n+1))
24
```

When it comes to parsing numbers, note that `float()`, `int()` and
`long()` all accept string arguments and will reject ill-formed strings by raising an `ValueError`. 
Using Backslash to Continue Statements

Since Python treats a newline as a statement terminator, and since statements are often more than is comfortable to put in one line, many people do:

```python
if foo.bar()['first'][0] == baz.quux(1, 2)[5:9] and \
calculate_number(10, 20) != forbulate(500, 360):
    pass
```

You should realize that this is dangerous: a stray space after the \ would make this line wrong, and stray spaces are notoriously hard to see in editors. In this case, at least it would be a syntax error, but if the code was:

```python
value = foo.bar()['first'][0]*baz.quux(1, 2)[5:9] \
      + calculate_number(10, 20)*forbulate(500, 360)
```

then it would just be subtly wrong.

It is usually much better to use the implicit continuation inside parenthesis:

This version is bulletproof:

```python
value = (foo.bar()['first'][0]*baz.quux(1, 2)[5:9] \
    + calculate_number(10, 20)*forbulate(500, 360))
```
In this document, we'll take a tour of Python’s features suitable for implementing programs in a functional style. After an introduction to the concepts of functional programming, we'll look at language features such as iterators and generators and relevant library modules such as itertools and functools.
This section explains the basic concept of functional programming; if you’re just interested in learning about Python language features, skip to the next section.

Programming languages support decomposing problems in several different ways:

- Most programming languages are **procedural**: programs are lists of instructions that tell the computer what to do with the program’s input. C, Pascal, and even Unix shells are procedural languages.
- In **declarative** languages, you write a specification that describes the problem to be solved, and the language implementation figures out how to perform the computation efficiently. SQL is the declarative language you’re most likely to be familiar with; a SQL query describes the data set you want to retrieve, and the SQL engine decides whether to scan tables or use indexes, which subclauses should be performed first, etc.
- **Object-oriented** programs manipulate collections of objects. Objects have internal state and support methods that query or modify this internal state in some way. Smalltalk and Java are object-oriented languages. C++ and Python are languages that support object-oriented programming, but don’t force the use of object-oriented features.
- **Functional** programming decomposes a problem into a set of functions. Ideally, functions only take inputs and produce outputs, and don’t have any internal state that affects the output produced for a given input. Well-known functional languages include the ML family (Standard ML, OCaml, and other variants) and Haskell.

The designers of some computer languages choose to emphasize
one particular approach to programming. This often makes it difficult to write programs that use a different approach. Other languages are multi-paradigm languages that support several different approaches. Lisp, C++, and Python are multi-paradigm; you can write programs or libraries that are largely procedural, object-oriented, or functional in all of these languages. In a large program, different sections might be written using different approaches; the GUI might be object-oriented while the processing logic is procedural or functional, for example.

In a functional program, input flows through a set of functions. Each function operates on its input and produces some output. Functional style discourages functions with side effects that modify internal state or make other changes that aren’t visible in the function’s return value. Functions that have no side effects at all are called purely functional. Avoiding side effects means not using data structures that get updated as a program runs; every function’s output must only depend on its input.

Some languages are very strict about purity and don’t even have assignment statements such as a=3 or c = a + b, but it’s difficult to avoid all side effects. Printing to the screen or writing to a disk file are side effects, for example. For example, in Python a print statement or a time.sleep(1) both return no useful value; they’re only called for their side effects of sending some text to the screen or pausing execution for a second.

Python programs written in functional style usually won’t go to the extreme of avoiding all I/O or all assignments; instead, they’ll provide a functional-appearing interface but will use non-functional features internally. For example, the implementation of a function will still use assignments to local variables, but won’t modify global variables or have other side effects.

Functional programming can be considered the opposite of object-
oriented programming. Objects are little capsules containing some internal state along with a collection of method calls that let you modify this state, and programs consist of making the right set of state changes. Functional programming wants to avoid state changes as much as possible and works with data flowing between functions. In Python you might combine the two approaches by writing functions that take and return instances representing objects in your application (e-mail messages, transactions, etc.).

Functional design may seem like an odd constraint to work under. Why should you avoid objects and side effects? There are theoretical and practical advantages to the functional style:

- Formal proviability.
- Modularity.
- Composability.
- Ease of debugging and testing.

**Formal proviability**

A theoretical benefit is that it's easier to construct a mathematical proof that a functional program is correct.

For a long time researchers have been interested in finding ways to mathematically prove programs correct. This is different from testing a program on numerous inputs and concluding that its output is usually correct, or reading a program’s source code and concluding that the code looks right; the goal is instead a rigorous proof that a program produces the right result for all possible inputs.

The technique used to prove programs correct is to write down invariants, properties of the input data and of the program’s variables that are always true. For each line of code, you then show that if invariants X and Y are true before the line is executed, the slightly different invariants X’ and Y’ are true after the line is
executed. This continues until you reach the end of the program, at which point the invariants should match the desired conditions on the program's output.

Functional programming's avoidance of assignments arose because assignments are difficult to handle with this technique; assignments can break invariants that were true before the assignment without producing any new invariants that can be propagated onward.

Unfortunately, proving programs correct is largely impractical and not relevant to Python software. Even trivial programs require proofs that are several pages long; the proof of correctness for a moderately complicated program would be enormous, and few or none of the programs you use daily (the Python interpreter, your XML parser, your web browser) could be proven correct. Even if you wrote down or generated a proof, there would then be the question of verifying the proof; maybe there's an error in it, and you wrongly believe you've proved the program correct.

Modularity

A more practical benefit of functional programming is that it forces you to break apart your problem into small pieces. Programs are more modular as a result. It's easier to specify and write a small function that does one thing than a large function that performs a complicated transformation. Small functions are also easier to read and to check for errors.

Ease of debugging and testing

Testing and debugging a functional-style program is easier.

Debugging is simplified because functions are generally small and clearly specified. When a program doesn't work, each function is an interface point where you can check that the data are correct. You
can look at the intermediate inputs and outputs to quickly isolate the function that’s responsible for a bug.

Testing is easier because each function is a potential subject for a unit test. Functions don’t depend on system state that needs to be replicated before running a test; instead you only have to synthesize the right input and then check that the output matches expectations.

Composability

As you work on a functional-style program, you’ll write a number of functions with varying inputs and outputs. Some of these functions will be unavoidably specialized to a particular application, but others will be useful in a wide variety of programs. For example, a function that takes a directory path and returns all the XML files in the directory, or a function that takes a filename and returns its contents, can be applied to many different situations.

Over time you'll form a personal library of utilities. Often you'll assemble new programs by arranging existing functions in a new configuration and writing a few functions specialized for the current task.
I’ll start by looking at a Python language feature that’s an important foundation for writing functional-style programs: iterators.

An iterator is an object representing a stream of data; this object returns the data one element at a time. A Python iterator must support a method called `next()` that takes no arguments and always returns the next element of the stream. If there are no more elements in the stream, `next()` must raise the `StopIteration` exception. Iterators don’t have to be finite, though; it’s perfectly reasonable to write an iterator that produces an infinite stream of data.

The built-in `iter()` function takes an arbitrary object and tries to return an iterator that will return the object’s contents or elements, raising `TypeError` if the object doesn’t support iteration. Several of Python’s built-in data types support iteration, the most common being lists and dictionaries. An object is called an `iterable` object if you can get an iterator for it.

You can experiment with the iteration interface manually:

```python
>>> L = [1,2,3]
>>> it = iter(L)
>>> print it
<...iterator object at ...>
>>> it.next()
1
>>> it.next()
2
>>> it.next()
3
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
```
Python expects iterable objects in several different contexts, the most important being the `for` statement. In the statement `for X in Y`, Y must be an iterator or some object for which `iter()` can create an iterator. These two statements are equivalent:

```python
for i in iter(obj):
    print i
for i in obj:
    print i
```

Iterators can be materialized as lists or tuples by using the `list()` or `tuple()` constructor functions:

```python
>>> L = [1,2,3]
>>> iterator = iter(L)
>>> t = tuple(iterator)
>>> t
(1, 2, 3)
```

Sequence unpacking also supports iterators: if you know an iterator will return N elements, you can unpack them into an N-tuple:

```python
>>> L = [1,2,3]
>>> iterator = iter(L)
>>> a,b,c = iterator
>>> a,b,c
(1, 2, 3)
```

Built-in functions such as `max()` and `min()` can take a single iterator argument and will return the largest or smallest element. The "in" and "not in" operators also support iterators: `X in iterator` is true if X is found in the stream returned by the iterator. You'll run into obvious problems if the iterator is infinite; `max()`, `min()` will never return, and if the element X never appears in the stream, the "in"
and "not in" operators won’t return either.

Note that you can only go forward in an iterator; there’s no way to get the previous element, reset the iterator, or make a copy of it. Iterator objects can optionally provide these additional capabilities, but the iterator protocol only specifies the `next()` method. Functions may therefore consume all of the iterator’s output, and if you need to do something different with the same stream, you’ll have to create a new iterator.

**Data Types That Support Iterators**

We’ve already seen how lists and tuples support iterators. In fact, any Python sequence type, such as strings, will automatically support creation of an iterator.

Calling `iter()` on a dictionary returns an iterator that will loop over the dictionary’s keys:

```python
c
>>> m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'Jun...
...     'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11, 'D
>>> for key in m:
...     print key, m[key]
Mar 3
Feb 2
Aug 8
Sep 9
Apr 4
Jun 6
Jul 7
Jan 1
May 5
Nov 11
Dec 12
Oct 10
```

Note that the order is essentially random, because it’s based on the hash ordering of the objects in the dictionary.
Applying `iter()` to a dictionary always loops over the keys, but dictionaries have methods that return other iterators. If you want to iterate over keys, values, or key/value pairs, you can explicitly call the `iterkeys()`, `itervalues()`, or `iteritems()` methods to get an appropriate iterator.

The `dict()` constructor can accept an iterator that returns a finite stream of (key, value) tuples:

```python
>>> L = [('Italy', 'Rome'), ('France', 'Paris'), ('US', 'Washington DC'), ('France', 'Paris')]
>>> dict(iter(L))
{'Italy': 'Rome', 'US': 'Washington DC', 'France': 'Paris'}
```

Files also support iteration by calling the `readline()` method until there are no more lines in the file. This means you can read each line of a file like this:

```python
for line in file:
    # do something for each line
...
```

Sets can take their contents from an iterable and let you iterate over the set's elements:

```python
S = set((2, 3, 5, 7, 11, 13))
for i in S:
    print i
```
Generator expressions and list comprehensions

Two common operations on an iterator’s output are 1) performing some operation for every element, 2) selecting a subset of elements that meet some condition. For example, given a list of strings, you might want to strip off trailing whitespace from each line or extract all the strings containing a given substring.

List comprehensions and generator expressions (short form: “listcomps” and “genexps”) are a concise notation for such operations, borrowed from the functional programming language Haskell (https://www.haskell.org/). You can strip all the whitespace from a stream of strings with the following code:

```python
line_list = [' line 1
', 'line 2 
', '...' ]

# Generator expression -- returns iterator
stripped_iter = (line.strip() for line in line_list)

# List comprehension -- returns list
stripped_list = [line.strip() for line in line_list]
```

You can select only certain elements by adding an "if" condition:

```python
stripped_list = [line.strip() for line in line_list
if line != '']
```

With a list comprehension, you get back a Python list; stripped_list is a list containing the resulting lines, not an iterator. Generator expressions return an iterator that computes the values as necessary, not needing to materialize all the values at once. This means that list comprehensions aren’t useful if you’re working with iterators that return an infinite stream or a very large amount of data. Generator expressions are preferable in these situations.
Generator expressions are surrounded by parentheses (“()”) and list comprehensions are surrounded by square brackets (“[]”). Generator expressions have the form:

```
(expression for expr in sequence1
    if condition1
    for expr2 in sequence2
    if condition2
    for expr3 in sequence3 ...
    if condition3
    for exprN in sequenceN
    if conditionN)
```

Again, for a list comprehension only the outside brackets are different (square brackets instead of parentheses).

The elements of the generated output will be the successive values of `expression`. The `if` clauses are all optional; if present, `expression` is only evaluated and added to the result when `condition` is true.

Generator expressions always have to be written inside parentheses, but the parentheses signalling a function call also count. If you want to create an iterator that will be immediately passed to a function you can write:

```
obj_total = sum(obj.count for obj in list_all_objects())
```

The `for...in` clauses contain the sequences to be iterated over. The sequences do not have to be the same length, because they are iterated over from left to right, **not** in parallel. For each element in `sequence1`, `sequence2` is looped over from the beginning. `sequence3` is then looped over for each resulting pair of elements from `sequence1` and `sequence2`.

To put it another way, a list comprehension or generator expression is equivalent to the following Python code:
for expr1 in sequence1:
    if not (condition1):
        continue  # Skip this element
for expr2 in sequence2:
    if not (condition2):
        continue  # Skip this element
...
for exprN in sequenceN:
    if not (conditionN):
        continue  # Skip this element

# Output the value of
# the expression.

This means that when there are multiple for...in clauses but no if clauses, the length of the resulting output will be equal to the product of the lengths of all the sequences. If you have two lists of length 3, the output list is 9 elements long:

```python
>>> seq1 = 'abc'
>>> seq2 = (1,2,3)
>>> [(x,y) for x in seq1 for y in seq2]
[('a', 1), ('a', 2), ('a', 3),
 ('b', 1), ('b', 2), ('b', 3),
 ('c', 1), ('c', 2), ('c', 3)]
```

To avoid introducing an ambiguity into Python’s grammar, if expression is creating a tuple, it must be surrounded with parentheses. The first list comprehension below is a syntax error, while the second one is correct:

```python
# Syntax error
[ x,y for x in seq1 for y in seq2]
# Correct
[ (x,y) for x in seq1 for y in seq2]
```
Generators

Generators are a special class of functions that simplify the task of writing iterators. Regular functions compute a value and return it, but generators return an iterator that returns a stream of values.

You’re doubtless familiar with how regular function calls work in Python or C. When you call a function, it gets a private namespace where its local variables are created. When the function reaches a `return` statement, the local variables are destroyed and the value is returned to the caller. A later call to the same function creates a new private namespace and a fresh set of local variables. But, what if the local variables weren’t thrown away on exiting a function? What if you could later resume the function where it left off? This is what generators provide; they can be thought of as resumable functions.

Here’s the simplest example of a generator function:

```python
def generate_ints(N):
    for i in range(N):
        yield i
```

Any function containing a `yield` keyword is a generator function; this is detected by Python’s `bytecode` compiler which compiles the function specially as a result.

When you call a generator function, it doesn’t return a single value; instead it returns a generator object that supports the iterator protocol. On executing the `yield` expression, the generator outputs the value of `i`, similar to a `return` statement. The big difference between `yield` and a `return` statement is that on reaching a `yield` the generator’s state of execution is suspended and local variables are preserved. On the next call to the generator’s `.next()` method, the function will resume executing.
Here's a sample usage of the `generate_ints()` generator:

```python
>>> gen = generate_ints(3)
>>> gen
<generator object generate_ints at ...>
>>> gen.next()
0
>>> gen.next()
1
>>> gen.next()
2
>>> gen.next()
Traceback (most recent call last):
  File "stdin", line 1, in ?
  File "stdin", line 2, in generate_ints
StopIteration
```

You could equally write `for i in generate_ints(5)`, or `a, b, c = generate_ints(3)`.

Inside a generator function, the `return` statement can only be used without a value, and signals the end of the procession of values; after executing a `return` the generator cannot return any further values. `return` with a value, such as `return 5`, is a syntax error inside a generator function. The end of the generator's results can also be indicated by raising `StopIteration` manually, or by just letting the flow of execution fall off the bottom of the function.

You could achieve the effect of generators manually by writing your own class and storing all the local variables of the generator as instance variables. For example, returning a list of integers could be done by setting `self.count` to 0, and having the `next()` method increment `self.count` and return it. However, for a moderately complicated generator, writing a corresponding class can be much messier.

The test suite included with Python's library, `test_generators.py`,
contains a number of more interesting examples. Here's one generator that implements an in-order traversal of a tree using generators recursively.

```python
# A recursive generator that generates Tree leaves in in-order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x

        yield t.label

        for x in inorder(t.right):
            yield x
```

Two other examples in `test_generators.py` produce solutions for the N-Queens problem (placing N queens on an NxN chess board so that no queen threatens another) and the Knight's Tour (finding a route that takes a knight to every square of an NxN chessboard without visiting any square twice).

### Passing values into a generator

In Python 2.4 and earlier, generators only produced output. Once a generator's code was invoked to create an iterator, there was no way to pass any new information into the function when its execution is resumed. You could hack together this ability by making the generator look at a global variable or by passing in some mutable object that callers then modify, but these approaches are messy.

In Python 2.5 there's a simple way to pass values into a generator. `yield` became an expression, returning a value that can be assigned to a variable or otherwise operated on:

```python
val = (yield i)
```
I recommend that you **always** put parentheses around a `yield` expression when you’re doing something with the returned value, as in the above example. The parentheses aren’t always necessary, but it’s easier to always add them instead of having to remember when they’re needed.

(PEP 342 explains the exact rules, which are that a `yield`-expression must always be parenthesized except when it occurs at the top-level expression on the right-hand side of an assignment. This means you can write `val = yield i` but have to use parentheses when there’s an operation, as in `val = (yield i) + 12`.)

Values are sent into a generator by calling its `send(value)` method. This method resumes the generator’s code and the `yield` expression returns the specified value. If the regular `next()` method is called, the `yield` returns `None`.

Here’s a simple counter that increments by 1 and allows changing the value of the internal counter.

```python
def counter (maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided, change counter
        if val is not None:
            i = val
        else:
            i += 1
```

And here’s an example of changing the counter:

```python
>>> it = counter(10)
>>> print it.next()
0
>>> print it.next()
1
>>> print it.send(8)
```
Because `yield` will often be returning `None`, you should always check for this case. Don’t just use its value in expressions unless you’re sure that the `send()` method will be the only method used to resume your generator function.

In addition to `send()`, there are two other new methods on generators:

- `throw(type, value=None, traceback=None)` is used to raise an exception inside the generator; the exception is raised by the `yield` expression where the generator’s execution is paused.

- `close()` raises a `GeneratorExit` exception inside the generator to terminate the iteration. On receiving this exception, the generator’s code must either raise `GeneratorExit` or `StopIteration`; catching the exception and doing anything else is illegal and will trigger a `RuntimeError`. `close()` will also be called by Python’s garbage collector when the generator is garbage-collected.

  If you need to run cleanup code when a `GeneratorExit` occurs, I suggest using a `try: ... finally:` suite instead of catching `GeneratorExit`.

The cumulative effect of these changes is to turn generators from one-way producers of information into both producers and consumers.
Generators also become **coroutines**, a more generalized form of subroutines. Subroutines are entered at one point and exited at another point (the top of the function, and a `return` statement), but coroutines can be entered, exited, and resumed at many different points (the `yield` statements).
Let's look in more detail at built-in functions often used with iterators.

Two of Python's built-in functions, `map()` and `filter()`, are somewhat obsolete; they duplicate the features of list comprehensions but return actual lists instead of iterators.

```
map(f, iterA, iterB, ...) returns a list containing f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), f(iterA[2], iterB[2]), ...
```

```sh
>>> def upper(s):
...     return s.upper()

>>> map(upper, ['sentence', 'fragment'])
['SENTENCE', 'FRAGMENT']

>>> [upper(s) for s in ['sentence', 'fragment']]
['SENTENCE', 'FRAGMENT']
```

As shown above, you can achieve the same effect with a list comprehension. The `itertools.imap()` function does the same thing but can handle infinite iterators; it'll be discussed later, in the section on the `itertools` module.

```
filter(predicate, iter) returns a list that contains all the sequence elements that meet a certain condition, and is similarly duplicated by list comprehensions. A `predicate` is a function that returns the truth value of some condition; for use with `filter()`, the predicate must take a single value.
```

```
>>> def is_even(x):
...     return (x % 2) == 0

>>> filter(is_even, range(10))
```
This can also be written as a list comprehension:

```python
>>> [x for x in range(10) if is_even(x)]
[0, 2, 4, 6, 8]
```

`filter()` also has a counterpart in the `itertools` module, `itertools.ifilter()`, that returns an iterator and can therefore handle infinite sequences just as `itertools.imap()` can.

`reduce(func, iter, [initial_value])` doesn’t have a counterpart in the `itertools` module because it cumulatively performs an operation on all the iterable’s elements and therefore can’t be applied to infinite iterables. `func` must be a function that takes two elements and returns a single value. `reduce()` takes the first two elements A and B returned by the iterator and calculates `func(A, B)`. It then requests the third element, C, calculates `func(func(A, B), C)`, combines this result with the fourth element returned, and continues until the iterable is exhausted. If the iterable returns no values at all, a `TypeError` exception is raised. If the initial value is supplied, it’s used as a starting point and `func(initial_value, A)` is the first calculation.

```python
>>> import operator
>>> reduce(operator.concat, ['A', 'BB', 'C'])
'ABBC'
>>> reduce(operator.concat, [])
Traceback (most recent call last):
  ...TypeError: reduce() of empty sequence with no initial value
```

If you use `operator.add()` with `reduce()`, you’ll add up all the
elements of the iterable. This case is so common that there's a special built-in called `sum()` to compute it:

```python
>>> reduce(operator.add, [1,2,3,4], 0)
10
>>> sum([1,2,3,4])
10
>>> sum([])
0
```

For many uses of `reduce()`, though, it can be clearer to just write the obvious `for` loop:

```python
# Instead of:
product = reduce(operator.mul, [1,2,3], 1)

# You can write:
product = 1
for i in [1,2,3]:
    product *= i
```

`enumerate(iter)` counts off the elements in the iterable, returning 2-tuples containing the count and each element.

```python
>>> for item in enumerate(['subject', 'verb', 'object']):
...     print item
(0, 'subject')
(1, 'verb')
(2, 'object')
```

`enumerate()` is often used when looping through a list and recording the indexes at which certain conditions are met:

```python
f = open('data.txt', 'r')
for i, line in enumerate(f):
    if line.strip() == '':
        print 'Blank line at line #%i' % i
```

```python
sorted(iterable, [cmp=None], [key=None], [reverse=False])
```
collects all the elements of the iterable into a list, sorts the list, and returns the sorted result. The \texttt{cmp}, \texttt{key}, and \texttt{reverse} arguments are passed through to the constructed list’s \texttt{.sort()} method.

```python
>>> import random
>>> # Generate 8 random numbers between [0, 10000)
>>> rand_list = random.sample(range(10000), 8)
>>> rand_list
[769, 7953, 9828, 6431, 8442, 9878, 6213, 2207]
>>> sorted(rand_list)
[769, 2207, 6213, 6431, 7953, 8442, 9828, 9878]
>>> sorted(rand_list, reverse=True)
[9878, 9828, 8442, 7953, 6431, 6213, 2207, 769]
```

(For a more detailed discussion of sorting, see the Sorting mini-HOWTO in the Python wiki at https://wiki.python.org/moin/HowTo/Sorting.)

The \texttt{any(iter)} and \texttt{all(iter)} built-ins look at the truth values of an iterable's contents. \texttt{any()} returns \texttt{True} if any element in the iterable is a true value, and \texttt{all()} returns \texttt{True} if all of the elements are true values:

```python
>>> any([0,1,0])
True
>>> any([0,0,0])
False
>>> any([1,1,1])
True
>>> all([0,1,0])
False
>>> all([0,0,0])
False
>>> all([1,1,1])
True
```
Small functions and the lambda expression

When writing functional-style programs, you'll often need little functions that act as predicates or that combine elements in some way.

If there's a Python built-in or a module function that's suitable, you don't need to define a new function at all:

```python
stripped_lines = [line.strip() for line in lines]
existing_files = filter(os.path.exists, file_list)
```

If the function you need doesn't exist, you need to write it. One way to write small functions is to use the `lambda` statement. `lambda` takes a number of parameters and an expression combining these parameters, and creates a small function that returns the value of the expression:

```python
lowercase = lambda x: x.lower()
print_assign = lambda name, value: name + '=' + str(value)
adder = lambda x, y: x+y
```

An alternative is to just use the `def` statement and define a function in the usual way:

```python
def lowercase(x):
    return x.lower()

def print_assign(name, value):
    return name + '=' + str(value)

def adder(x,y):
    return x + y
```

Which alternative is preferable? That's a style question; my usual
course is to avoid using \texttt{lambda}.

One reason for my preference is that \texttt{lambda} is quite limited in the functions it can define. The result has to be computable as a single expression, which means you can’t have multiway if... elif... else comparisons or try... except statements. If you try to do too much in a \texttt{lambda} statement, you’ll end up with an overly complicated expression that’s hard to read. Quick, what’s the following code doing?

\begin{verbatim}
total = reduce(lambda a, b: (0, a[1] + b[1]), items)[1]
\end{verbatim}

You can figure it out, but it takes time to disentangle the expression to figure out what’s going on. Using a short nested \texttt{def} statements makes things a little bit better:

\begin{verbatim}
def combine (a, b):
    return (0, a[1] + b[1])
total = reduce(combine, items)[1]
\end{verbatim}

But it would be best of all if I had simply used a \texttt{for} loop:

\begin{verbatim}
total = 0
for a, b in items:
    total += b
\end{verbatim}

Or the \texttt{sum()} built-in and a generator expression:

\begin{verbatim}
total = sum(b for a,b in items)
\end{verbatim}

Many uses of \texttt{reduce()} are clearer when written as \texttt{for} loops.

Fredrik Lundh once suggested the following set of rules for refactoring uses of \texttt{lambda}:
1. Write a lambda function.
2. Write a comment explaining what the heck that lambda does.
3. Study the comment for a while, and think of a name that captures the essence of the comment.
4. Convert the lambda to a def statement, using that name.
5. Remove the comment.

I really like these rules, but you’re free to disagree about whether this lambda-free style is better.
The *itertools* module contains a number of commonly-used iterators as well as functions for combining several iterators. This section will introduce the module's contents by showing small examples.

The module's functions fall into a few broad classes:

- Functions that create a new iterator based on an existing iterator.
- Functions for treating an iterator’s elements as function arguments.
- Functions for selecting portions of an iterator’s output.
- A function for grouping an iterator’s output.

**Creating new iterators**

*itertools.count(n)* returns an infinite stream of integers, increasing by 1 each time. You can optionally supply the starting number, which defaults to 0:

```python
itertools.count() =>
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...

tools.count(10) =>
  10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...
```

*itertools.cycle(iter)* saves a copy of the contents of a provided iterable and returns a new iterator that returns its elements from first to last. The new iterator will repeat these elements infinitely.

```python
itertools.cycle([1,2,3,4,5]) =>
  1, 2, 3, 4, 5, 1, 2, 3, 4, 5, ...
```
`itertools.repeat(elem, [n])` returns the provided element \( n \) times, or returns the element endlessly if \( n \) is not provided.

```python
itertools.repeat('abc') =>
  abc, abc, abc, abc, abc, abc, abc, abc, abc, ...
itertools.repeat('abc', 5) =>
  abc, abc, abc, abc, abc
```

`itertools.chain(iterA, iterB, ...)` takes an arbitrary number of iterables as input, and returns all the elements of the first iterator, then all the elements of the second, and so on, until all of the iterables have been exhausted.

```python
itertools.chain(["a", "b", "c"], (1, 2, 3)) =>
  a, b, c, 1, 2, 3
```

`itertools.izip(iterA, iterB, ...)` takes one element from each iterable and returns them in a tuple:

```python
itertools.izip(["a", "b", "c"], (1, 2, 3)) =>
  ("a", 1), ("b", 2), ("c", 3)
```

It's similar to the built-in `zip()` function, but doesn't construct an in-memory list and exhaust all the input iterators before returning; instead tuples are constructed and returned only if they're requested. (The technical term for this behaviour is lazy evaluation.)

This iterator is intended to be used with iterables that are all of the same length. If the iterables are of different lengths, the resulting stream will be the same length as the shortest iterable.

```python
itertools.izip(["a", "b"], (1, 2, 3)) =>
  ("a", 1), ("b", 2)
```

You should avoid doing this, though, because an element may be taken from the longer iterators and discarded. This means you can't
go on to use the iterators further because you risk skipping a discarded element.

```
import itertools

# itertools.islice(iter, [start], stop, [step]) returns a stream that's a slice of the iterator. With a single stop argument, it will return the first stop elements. If you supply a starting index, you'll get stop-start elements, and if you supply a value for step, elements will be skipped accordingly. Unlike Python's string and list slicing, you can't use negative values for start, stop, or step.

itertools.islice(range(10), 8) =>
  0, 1, 2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8) =>
  2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8, 2) =>
  2, 4, 6
```

```
# itertools.tee(iter, [n]) replicates an iterator; it returns n independent iterators that will all return the contents of the source iterator. If you don't supply a value for n, the default is 2. Replicating iterators requires saving some of the contents of the source iterator, so this can consume significant memory if the iterator is large and one of the new iterators is consumed more than the others.

itertools.tee(itertools.count()) =>
  iterA, iterB
where iterA ->
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
and iterB ->
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
```

Calling functions on elements

Two functions are used for calling other functions on the contents of an iterable.
itertools.imap(f, iterA, iterB, ...) returns a stream containing
f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), f(iterA[2], iterB[2]), ...:

```
import operator

itertools.imap(operator.add, [5, 6, 5], [1, 2, 3]) =>
6, 8, 8
```

The `operator` module contains a set of functions corresponding to
Python's operators. Some examples are `operator.add(a, b)` (adds
two values), `operator.ne(a, b)` (same as `a!=b`), and
`operator.attrgetter('id')` (returns a callable that fetches the "id"
attribute).

```
def is_even(x):
    return (x % 2) == 0

itertools.ifilter(is_even, itertools.count()) =>
0, 2, 4, 6, 8, 10, 12, 14, ...
```
**itertools.ifilterfalse(predicate, iter)** is the opposite, returning all elements for which the predicate returns false:

```python
itertools.ifilterfalse(is_even, itertools.count()) =>
    1, 3, 5, 7, 9, 11, 13, 15, ...
```

**itertools.takewhile(predicate, iter)** returns elements for as long as the predicate returns true. Once the predicate returns false, the iterator will signal the end of its results.

```python
def less_than_10(x):
    return (x < 10)

itertools.takewhile(less_than_10, itertools.count()) =>
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9

itertools.takewhile(is_even, itertools.count()) =>
    0
```

**itertools.dropwhile(predicate, iter)** discards elements while the predicate returns true, and then returns the rest of the iterable’s results.

```python
itertools.dropwhile(less_than_10, itertools.count()) =>
    10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...

itertools.dropwhile(is_even, itertools.count()) =>
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
```

### Grouping elements

The last function I’ll discuss, **itertools.groupby(iter, key_func=None)**, is the most complicated. **key_func(elem)** is a function that can compute a key value for each element returned by the iterable. If you don’t supply a key function, the key is simply each element itself.
`groupby()` collects all the consecutive elements from the underlying iterable that have the same key value, and returns a stream of 2-tuples containing a key value and an iterator for the elements with that key.

```python
city_list = [('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL'), ('Anchorage', 'AK'), ('Nome', 'AK'), ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ'), ...
```

def get_state((city, state)):
    return state

itertools.groupby(city_list, get_state) =>
    ('AL', iterator-1),
    ('AK', iterator-2),
    ('AZ', iterator-3), ...

where
iterator-1 =>
    ('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL')
iterator-2 =>
    ('Anchorage', 'AK'), ('Nome', 'AK')
iterator-3 =>
    ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ')
```

`groupby()` assumes that the underlying iterable’s contents will already be sorted based on the key. Note that the returned iterators also use the underlying iterable, so you have to consume the results of iterator-1 before requesting iterator-2 and its corresponding key.
The **functools** module in Python 2.5 contains some higher-order functions. A **higher-order function** takes one or more functions as input and returns a new function. The most useful tool in this module is the **functools.partial()** function.

For programs written in a functional style, you'll sometimes want to construct variants of existing functions that have some of the parameters filled in. Consider a Python function `f(a, b, c);` you may wish to create a new function `g(b, c)` that's equivalent to `f(1, b, c);` you're filling in a value for one of `f()`'s parameters. This is called "partial function application".

The constructor for **partial** takes the arguments `(function, arg1, arg2, ... kwarg1=value1, kwarg2=value2)`. The resulting object is callable, so you can just call it to invoke `function` with the filled-in arguments.

Here's a small but realistic example:

```python
import functools

def log (message, subsystem):
    "Write the contents of 'message' to the specified subsystem"
    print '%s: %s' % (subsystem, message)
...

server_log = functools.partial(log, subsystem='server')
server_log('Unable to open socket')
```

The **operator** module was mentioned earlier. It contains a set of
functions corresponding to Python’s operators. These functions are often useful in functional-style code because they save you from writing trivial functions that perform a single operation.

Some of the functions in this module are:

- Math operations: `add()`, `sub()`, `mul()`, `div()`, `floordiv()`, `abs()`, ...
- Logical operations: `not()``, `truth()`.
- Bitwise operations: `and()`, `or()`, `invert()`.
- Comparisons: `eq()`, `ne()`, `lt()`, `le()`, `gt()`, and `ge()`.
- Object identity: `is()``, `is_not()`.

Consult the operator module’s documentation for a complete list.
Revision History and Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Ian Bicking, Nick Coghlan, Nick Efford, Raymond Hettinger, Jim Jewett, Mike Krell, Leandro Lameiro, Jussi Salmela, Collin Winter, Blake Winton.

Version 0.1: posted June 30 2006.

Version 0.11: posted July 1 2006. Typo fixes.

Version 0.2: posted July 10 2006. Merged genexp and listcomp sections into one. Typo fixes.

Version 0.21: Added more references suggested on the tutor mailing list.

Version 0.30: Adds a section on the functional module written by Collin Winter; adds short section on the operator module; a few other edits.
**General**

**Structure and Interpretation of Computer Programs**, by Harold Abelson and Gerald Jay Sussman with Julie Sussman. Full text at [https://mitpress.mit.edu/sicp/](https://mitpress.mit.edu/sicp/). In this classic textbook of computer science, chapters 2 and 3 discuss the use of sequences and streams to organize the data flow inside a program. The book uses Scheme for its examples, but many of the design approaches described in these chapters are applicable to functional-style Python code.

[http://www.defmacro.org/ramblings/fp.html](http://www.defmacro.org/ramblings/fp.html): A general introduction to functional programming that uses Java examples and has a lengthy historical introduction.


**Python-specific**


Mertz also wrote a 3-part series of articles on functional programming for IBM’s DeveloperWorks site; see
part 1, part 2, and part 3,

Python documentation

Documentation for the *itertools* module.

Documentation for the *operator* module.

**PEP 289**: “Generator Expressions”

**PEP 342**: “Coroutines via Enhanced Generators” describes the new generator features in Python 2.5.
Basic Logging Tutorial

Logging is a means of tracking events that happen when some software runs. The software's developer adds logging calls to their code to indicate that certain events have occurred. An event is described by a descriptive message which can optionally contain variable data (i.e. data that is potentially different for each occurrence of the event). Events also have an importance which the developer ascribes to the event; the importance can also be called the level or severity.

When to use logging

Logging provides a set of convenience functions for simple logging usage. These are `debug()`, `info()`, `warning()`, `error()` and `critical()`. To determine when to use logging, see the table below, which states, for each of a set of common tasks, the best tool to use for it.

<table>
<thead>
<tr>
<th>Task you want to perform</th>
<th>The best tool for the task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display console output for ordinary usage of a command line script or program</td>
<td><code>print()</code></td>
</tr>
<tr>
<td>Report events that occur during normal operation of a program (e.g. for status monitoring or fault investigation)</td>
<td><code>logging.info()</code> (or <code>logging.debug()</code> for very detailed output for diagnostic purposes)</td>
</tr>
<tr>
<td>Issue a warning regarding a particular runtime event</td>
<td><code>warnings.warn()</code> in library code if the issue is avoidable and the client application should be modified to eliminate the warning</td>
</tr>
<tr>
<td></td>
<td><code>logging.warning()</code> if there is</td>
</tr>
</tbody>
</table>
nothing the client application can do about the situation, but the event should still be noted

<table>
<thead>
<tr>
<th>Report an error regarding a particular runtime event</th>
<th>Raise an exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report suppression of an error without raising an exception (e.g. error handler in a long-running server process)</td>
<td><code>logging.error()</code>, <code>logging.exception()</code> or <code>logging.critical()</code> as appropriate for the specific error and application domain</td>
</tr>
</tbody>
</table>

The logging functions are named after the level or severity of the events they are used to track. The standard levels and their applicability are described below (in increasing order of severity):

<table>
<thead>
<tr>
<th>Level</th>
<th>When it’s used</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG</td>
<td>Detailed information, typically of interest only when diagnosing problems.</td>
</tr>
<tr>
<td>INFO</td>
<td>Confirmation that things are working as expected.</td>
</tr>
<tr>
<td>WARNING</td>
<td>An indication that something unexpected happened, or indicative of some problem in the near future (e.g. ‘disk space low’). The software is still working as expected.</td>
</tr>
<tr>
<td>ERROR</td>
<td>Due to a more serious problem, the software has not been able to perform some function.</td>
</tr>
<tr>
<td>CRITICAL</td>
<td>A serious error, indicating that the program itself may be unable to continue running.</td>
</tr>
</tbody>
</table>

The default level is **WARNING**, which means that only events of this level and above will be tracked, unless the logging package is configured to do otherwise.

Events that are tracked can be handled in different ways. The simplest way of handling tracked events is to print them to the console. Another common way is to write them to a disk file.
**A simple example**

A very simple example is:

```python
import logging
logging.warning('Watch out!')  # will print a message to the console
logging.info('I told you so')  # will not print anything
```

If you type these lines into a script and run it, you'll see:

```
WARNING:root:Watch out!
```

printed out on the console. The `INFO` message doesn’t appear because the default level is `WARNING`. The printed message includes the indication of the level and the description of the event provided in the logging call, i.e. ‘Watch out!’. Don’t worry about the ‘root’ part for now: it will be explained later. The actual output can be formatted quite flexibly if you need that; formatting options will also be explained later.

**Logging to a file**

A very common situation is that of recording logging events in a file, so let’s look at that next. Be sure to try the following in a newly-started Python interpreter, and don’t just continue from the session described above:

```python
import logging
logging.basicConfig(filename='example.log', level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
```

And now if we open the file and look at what we have, we should find the log messages:
This example also shows how you can set the logging level which acts as the threshold for tracking. In this case, because we set the threshold to DEBUG, all of the messages were printed.

If you want to set the logging level from a command-line option such as:

```bash
--log=INFO
```

and you have the value of the parameter passed for `--log` in some variable `loglevel`, you can use:

```python
getattr(logging, loglevel.upper())
```

to get the value which you’ll pass to `basicConfig()` via the `level` argument. You may want to error check any user input value, perhaps as in the following example:

```python
# assuming loglevel is bound to the string value obtained from # command line argument. Convert to upper case to allow the use # specify --log=DEBUG or --log=debug numeric_level = getattr(logging, loglevel.upper(), None) if not isinstance(numeric_level, int):
    raise ValueError('Invalid log level: %s' % loglevel) logging.basicConfig(level=numeric_level, ...)
```

The call to `basicConfig()` should come before any calls to `debug()`, `info()` etc. As it’s intended as a one-off simple configuration facility, only the first call will actually do anything: subsequent calls are effectively no-ops.

If you run the above script several times, the messages from
successive runs are appended to the file *example.log*. If you want each run to start afresh, not remembering the messages from earlier runs, you can specify the *filemode* argument, by changing the call in the above example to:

```
logging.basicConfig(filename='example.log', filemode='w', level=logging.INFO)
```

The output will be the same as before, but the log file is no longer appended to, so the messages from earlier runs are lost.

## Logging from multiple modules

If your program consists of multiple modules, here’s an example of how you could organize logging in it:

```
# myapp.py
import logging
import mylib

def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logging.info('Started')
    mylib.do_something()
    logging.info('Finished')

if __name__ == '__main__':
    main()
```

```
# mylib.py
import logging

def do_something():
    logging.info('Doing something')
```

If you run *myapp.py*, you should see this in *myapp.log*:

```
INFO:root:Started
INFO:root:Doing something
```
which is hopefully what you were expecting to see. You can
generalize this to multiple modules, using the pattern in mylib.py.
Note that for this simple usage pattern, you won't know, by looking in
the log file, where in your application your messages came from, 
apart from looking at the event description. If you want to track the
location of your messages, you'll need to refer to the documentation
beyond the tutorial level – see Advanced Logging Tutorial.

Logging variable data

To log variable data, use a format string for the event description
message and append the variable data as arguments. For example:

```python
import logging
logging.warning('%s before you %s', 'Look', 'leap!')
```

will display:

```
WARNING:root:Look before you leap!
```

As you can see, merging of variable data into the event description
message uses the old, %-style of string formatting. This is for
backwards compatibility: the logging package pre-dates newer
formatting options such as `str.format()` and `string.Template`.
These newer formatting options are supported, but exploring them is
outside the scope of this tutorial.

Changing the format of displayed messages

To change the format which is used to display messages, you need
to specify the format you want to use:

```python
import logging
```
logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)
logging.debug('This message should appear on the console')
logging.info('So should this')
logging.warning('And this, too')

which would print:

DEBUG:This message should appear on the console
INFO:So should this
WARNING:And this, too

Notice that the ‘root’ which appeared in earlier examples has disappeared. For a full set of things that can appear in format strings, you can refer to the documentation for LogRecord attributes, but for simple usage, you just need thelevelname(severity), message(event description, including variable data) and perhaps to display when the event occurred. This is described in the next section.

Displaying the date/time in messages

To display the date and time of an event, you would place ‘% (asctime)s’ in your format string:

import logging
logging.basicConfig(format='%s %s', datefmt='%Y-%m-%d %H:%M:%S')
logging.warning('is when this event was logged.')

which should print something like this:

2010-12-12 11:41:42,612 is when this event was logged.

The default format for date/time display (shown above) is ISO8601. If you need more control over the formatting of the date/time, provide a datefmt argument to basicConfig, as in this example:
logging.warning('is when this event was logged.')

which would display something like this:

12/12/2010 11:46:36 AM is when this event was logged.

The format of the datefmt argument is the same as supported by time.strftime().

Next Steps

That concludes the basic tutorial. It should be enough to get you up and running with logging. There’s a lot more that the logging package offers, but to get the best out of it, you’ll need to invest a little more of your time in reading the following sections. If you’re ready for that, grab some of your favourite beverage and carry on.

If your logging needs are simple, then use the above examples to incorporate logging into your own scripts, and if you run into problems or don’t understand something, please post a question on the comp.lang.python Usenet group (available at https://groups.google.com/group/comp.lang.python) and you should receive help before too long.

Still here? You can carry on reading the next few sections, which provide a slightly more advanced/in-depth tutorial than the basic one above. After that, you can take a look at the Logging Cookbook.
Advanced Logging Tutorial

The logging library takes a modular approach and offers several categories of components: loggers, handlers, filters, and formatters.

- Loggers expose the interface that application code directly uses.
- Handlers send the log records (created by loggers) to the appropriate destination.
- Filters provide a finer grained facility for determining which log records to output.
- Formatters specify the layout of log records in the final output.

Log event information is passed between loggers, handlers, filters and formatters in a `LogRecord` instance.

Logging is performed by calling methods on instances of the `Logger` class (hereafter called `loggers`). Each instance has a name, and they are conceptually arranged in a namespace hierarchy using dots (periods) as separators. For example, a logger named ‘scan’ is the parent of loggers ‘scan.text’, ‘scan.html’ and ‘scan.pdf’. Logger names can be anything you want, and indicate the area of an application in which a logged message originates.

A good convention to use when naming loggers is to use a module-level logger, in each module which uses logging, named as follows:

```python
logger = logging.getLogger(__name__)
```

This means that logger names track the package/module hierarchy, and it’s intuitively obvious where events are logged just from the logger name.

The root of the hierarchy of loggers is called the root logger. That’s the logger used by the functions `debug()`, `info()`, `warning()`, `error()`
and **critical()**, which just call the same-named method of the root logger. The functions and the methods have the same signatures. The root logger’s name is printed as ‘root’ in the logged output.

It is, of course, possible to log messages to different destinations. Support is included in the package for writing log messages to files, HTTP GET/POST locations, email via SMTP, generic sockets, or OS-specific logging mechanisms such as syslog or the Windows NT event log. Destinations are served by *handler* classes. You can create your own log destination class if you have special requirements not met by any of the built-in handler classes.

By default, no destination is set for any logging messages. You can specify a destination (such as console or file) by using `basicConfig()` as in the tutorial examples. If you call the functions `debug()`, `info()`, `warning()`, `error()` and `critical()`, they will check to see if no destination is set; and if one is not set, they will set a destination of the console (`sys.stderr`) and a default format for the displayed message before delegating to the root logger to do the actual message output.

The default format set by `basicConfig()` for messages is:

```
severity:logger name:message
```

You can change this by passing a format string to `basicConfig()` with the *format* keyword argument. For all options regarding how a format string is constructed, see [Formatter Objects](#).

---

**Logging Flow**

The flow of log event information in loggers and handlers is illustrated in the following diagram.
Loggers

Logger objects have a threelfold job. First, they expose several methods to application code so that applications can log messages at runtime. Second, logger objects determine which log messages to act upon based upon severity (the default filtering facility) or filter objects. Third, logger objects pass along relevant log messages to all interested log handlers.

The most widely used methods on logger objects fall into two categories: configuration and message sending.

These are the most common configuration methods:
• **Logger.setLevel()** specifies the lowest-severity log message a logger will handle, where debug is the lowest built-in severity level and critical is the highest built-in severity. For example, if the severity level is INFO, the logger will handle only INFO, WARNING, ERROR, and CRITICAL messages and will ignore DEBUG messages.

• **Logger.addHandler()** and **Logger.removeHandler()** add and remove handler objects from the logger object. Handlers are covered in more detail in [Handlers](#).

• **Logger.addFilter()** and **Logger.removeFilter()** add and remove filter objects from the logger object. Filters are covered in more detail in [Filter Objects](#).

You don’t need to always call these methods on every logger you create. See the last two paragraphs in this section.

With the logger object configured, the following methods create log messages:

• **Logger.debug(), Logger.info(), Logger.warning(), Logger.error(), and Logger.critical()** all create log records with a message and a level that corresponds to their respective method names. The message is actually a format string, which may contain the standard string substitution syntax of %s, %d, %f, and so on. The rest of their arguments is a list of objects that correspond with the substitution fields in the message. With regard to **kwargs, the logging methods care only about a keyword of exc_info and use it to determine whether to log exception information.

• **Logger.exception()** creates a log message similar to **Logger.error()**. The difference is that **Logger.exception()** dumps a stack trace along with it. Call this method only from an exception handler.

• **Logger.log()** takes a log level as an explicit argument. This is a
little more verbose for logging messages than using the log level convenience methods listed above, but this is how to log at custom log levels.

`getLogger()` returns a reference to a logger instance with the specified name if it is provided, or `root` if not. The names are period-separated hierarchical structures. Multiple calls to `getLogger()` with the same name will return a reference to the same logger object. Loggers that are further down in the hierarchical list are children of loggers higher up in the list. For example, given a logger with a name of `foo`, loggers with names of `foo.bar`, `foo.bar.baz`, and `foo.bam` are all descendants of `foo`.

Loggers have a concept of *effective level*. If a level is not explicitly set on a logger, the level of its parent is used instead as its effective level. If the parent has no explicit level set, its parent is examined, and so on - all ancestors are searched until an explicitly set level is found. The root logger always has an explicit level set (`WARNING` by default). When deciding whether to process an event, the effective level of the logger is used to determine whether the event is passed to the logger’s handlers.

Child loggers propagate messages up to the handlers associated with their ancestor loggers. Because of this, it is unnecessary to define and configure handlers for all the loggers an application uses. It is sufficient to configure handlers for a top-level logger and create child loggers as needed. (You can, however, turn off propagation by setting the `propagate` attribute of a logger to `False`.)

**Handlers**

Handler objects are responsible for dispatching the appropriate log messages (based on the log messages’ severity) to the handler’s specified destination. Logger objects can add zero or more handler
objects to themselves with an `addHandler()` method. As an example scenario, an application may want to send all log messages to a log file, all log messages of error or higher to stdout, and all messages of critical to an email address. This scenario requires three individual handlers where each handler is responsible for sending messages of a specific severity to a specific location.

The standard library includes quite a few handler types (see Useful Handlers); the tutorials use mainly `StreamHandler` and `FileHandler` in its examples.

There are very few methods in a handler for application developers to concern themselves with. The only handler methods that seem relevant for application developers who are using the built-in handler objects (that is, not creating custom handlers) are the following configuration methods:

- The `setLevel()` method, just as in logger objects, specifies the lowest severity that will be dispatched to the appropriate destination. Why are there two `setLevel()` methods? The level set in the logger determines which severity of messages it will pass to its handlers. The level set in each handler determines which messages that handler will send on.
- `setFormatter()` selects a Formatter object for this handler to use.
- `addFilter()` and `removeFilter()` respectively configure and deconfigure filter objects on handlers.

Application code should not directly instantiate and use instances of `Handler`. Instead, the `Handler` class is a base class that defines the interface that all handlers should have and establishes some default behavior that child classes can use (or override).

---

**Formatters**
Formatter objects configure the final order, structure, and contents of the log message. Unlike the base `logging.Handler` class, application code may instantiate formatter classes, although you could likely subclass the formatter if your application needs special behavior. The constructor takes two optional arguments – a message format string and a date format string.

```
logging.Formatter.__init__(fmt=None, datefmt=None)
```

If there is no message format string, the default is to use the raw message. If there is no date format string, the default date format is:

```
%Y-%m-%d %H:%M:%S
```

with the milliseconds tacked on at the end.

The message format string uses `%(<dictionary key>)s` styled string substitution; the possible keys are documented in `LogRecord attributes`.

The following message format string will log the time in a human-readable format, the severity of the message, and the contents of the message, in that order:

```
'%(asctime)s - %(levelname)s - %(message)s'
```

Formatters use a user-configurable function to convert the creation time of a record to a tuple. By default, `time.localtime()` is used; to change this for a particular formatter instance, set the `converter` attribute of the instance to a function with the same signature as `time.localtime()` or `time.gmtime()`. To change it for all formatters, for example if you want all logging times to be shown in GMT, set the `converter` attribute in the Formatter class (to `time.gmtime` for GMT display).
Configuring Logging

Programmers can configure logging in three ways:

1. Creating loggers, handlers, and formatters explicitly using Python code that calls the configuration methods listed above.
2. Creating a logging config file and reading it using the `fileConfig()` function.
3. Creating a dictionary of configuration information and passing it to the `dictConfig()` function.

For the reference documentation on the last two options, see `Configuration functions`. The following example configures a very simple logger, a console handler, and a simple formatter using Python code:

```python
import logging

# create logger
logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)

# create console handler and set level to debug
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)

# create formatter
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# add formatter to ch
ch.setFormatter(formatter)

# add ch to logger
logger.addHandler(ch)

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```
Running this module from the command line produces the following output:

$ python simple_logging_module.py
2005-03-19 15:10:26,618 - simple_example - DEBUG - debug message
2005-03-19 15:10:26,620 - simple_example - INFO - info message
2005-03-19 15:10:26,695 - simple_example - WARNING - warn message
2005-03-19 15:10:26,697 - simple_example - ERROR - error message
2005-03-19 15:10:26,773 - simple_example - CRITICAL - critical message

The following Python module creates a logger, handler, and formatter nearly identical to those in the example listed above, with the only difference being the names of the objects:

```python
ing
import logging
import logging.config

logging.config.fileConfig('logging.conf')

# create logger
logger = logging.getLogger('simpleExample')

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

Here is the logging.conf file:

```
[loggers]
keys=root,simpleExample

[handlers]
keys=consoleHandler

[formatters]
keys=simpleFormatter
```
The output is nearly identical to that of the non-config-file-based example:

```
$ python simple_logging_config.py
2005-03-19 15:38:55,977 - simpleExample - DEBUG - debug message
2005-03-19 15:38:55,979 - simpleExample - INFO - info message
2005-03-19 15:38:56,054 - simpleExample - WARNING - warn message
2005-03-19 15:38:56,055 - simpleExample - ERROR - error message
2005-03-19 15:38:56,130 - simpleExample - CRITICAL - critical message
```

You can see that the config file approach has a few advantages over the Python code approach, mainly separation of configuration and code and the ability of noncoders to easily modify the logging properties.

**Warning:** The `fileConfig()` function takes a default parameter, `disable_existing_loggers`, which defaults to `True` for reasons of backward compatibility. This may or may not be what you want, since it will cause any loggers existing before the `fileConfig()` call to be disabled unless they (or an ancestor) are explicitly named in
the configuration. Please refer to the reference documentation for more information, and specify False for this parameter if you wish.

The dictionary passed to dictConfig() can also specify a Boolean value with key disable_existing_loggers, which if not specified explicitly in the dictionary also defaults to being interpreted as True. This leads to the logger-disabling behaviour described above, which may not be what you want - in which case, provide the key explicitly with a value of False.

Note that the class names referenced in config files need to be either relative to the logging module, or absolute values which can be resolved using normal import mechanisms. Thus, you could use either WatchedFileHandler (relative to the logging module) or mypackage.mymodule.MyHandler (for a class defined in package mypackage and module mymodule, where mypackage is available on the Python import path).

In Python 2.7, a new means of configuring logging has been introduced, using dictionaries to hold configuration information. This provides a superset of the functionality of the config-file-based approach outlined above, and is the recommended configuration method for new applications and deployments. Because a Python dictionary is used to hold configuration information, and since you can populate that dictionary using different means, you have more options for configuration. For example, you can use a configuration file in JSON format, or, if you have access to YAML processing functionality, a file in YAML format, to populate the configuration dictionary. Or, of course, you can construct the dictionary in Python code, receive it in pickled form over a socket, or use whatever approach makes sense for your application.

Here's an example of the same configuration as above, in YAML format for the new dictionary-based approach:
For more information about logging using a dictionary, see Configuration functions.

**What happens if no configuration is provided**

If no logging configuration is provided, it is possible to have a situation where a logging event needs to be output, but no handlers can be found to output the event. The behaviour of the logging package in these circumstances is dependent on the Python version.

For Python 2.x, the behaviour is as follows:

- If `logging.raiseExceptions` is `False` (production mode), the event is silently dropped.
- If `logging.raiseExceptions` is `True` (development mode), a message ‘No handlers could be found for logger X.Y.Z’ is printed once.

**Configuring Logging for a Library**
When developing a library which uses logging, you should take care to document how the library uses logging - for example, the names of loggers used. Some consideration also needs to be given to its logging configuration. If the using application does not configure logging, and library code makes logging calls, then (as described in the previous section) an error message will be printed to \texttt{sys.stderr}.

If for some reason you \textit{don't} want this message printed in the absence of any logging configuration, you can attach a do-nothing handler to the top-level logger for your library. This avoids the message being printed, since a handler will be always be found for the library's events: it just doesn't produce any output. If the library user configures logging for application use, presumably that configuration will add some handlers, and if levels are suitably configured then logging calls made in library code will send output to those handlers, as normal.

A do-nothing handler is included in the logging package: \texttt{NullHandler} (since Python 2.7). An instance of this handler could be added to the top-level logger of the logging namespace used by the library \textit{(if you want to prevent an error message being output to \texttt{sys.stderr} in the absence of logging configuration)}. If all logging by a library \texttt{foo} is done using loggers with names matching `foo.x', `foo.x.y', etc. then the code:

```python
import logging
logging.getLogger('foo').addHandler(logging.NullHandler())
```

should have the desired effect. If an organisation produces a number of libraries, then the logger name specified can be `orgname.foo' rather than just `foo'.

\textbf{Note:} It is strongly advised that you \textit{do not add any handlers other than NullHandler} to your library's loggers. This is because the configuration of handlers is the prerogative of the application.
developer who uses your library. The application developer knows their target audience and what handlers are most appropriate for their application: if you add handlers ‘under the hood’, you might well interfere with their ability to carry out unit tests and deliver logs which suit their requirements.
Logging Levels

The numeric values of logging levels are given in the following table. These are primarily of interest if you want to define your own levels, and need them to have specific values relative to the predefined levels. If you define a level with the same numeric value, it overwrites the predefined value; the predefined name is lost.

<table>
<thead>
<tr>
<th>Level</th>
<th>Numeric value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRITICAL</td>
<td>50</td>
</tr>
<tr>
<td>ERROR</td>
<td>40</td>
</tr>
<tr>
<td>WARNING</td>
<td>30</td>
</tr>
<tr>
<td>INFO</td>
<td>20</td>
</tr>
<tr>
<td>DEBUG</td>
<td>10</td>
</tr>
<tr>
<td>NOTSET</td>
<td>0</td>
</tr>
</tbody>
</table>

Levels can also be associated with loggers, being set either by the developer or through loading a saved logging configuration. When a logging method is called on a logger, the logger compares its own level with the level associated with the method call. If the logger's level is higher than the method call's, no logging message is actually generated. This is the basic mechanism controlling the verbosity of logging output.

Logging messages are encoded as instances of the LogRecord class. When a logger decides to actually log an event, a LogRecord instance is created from the logging message.

Logging messages are subjected to a dispatch mechanism through the use of handlers, which are instances of subclasses of the Handler class. Handlers are responsible for ensuring that a logged message (in the form of a LogRecord) ends up in a particular location.
(or set of locations) which is useful for the target audience for that message (such as end users, support desk staff, system administrators, developers). Handlers are passed LogRecord instances intended for particular destinations. Each logger can have zero, one or more handlers associated with it (via the addHandler() method of Logger). In addition to any handlers directly associated with a logger, all handlers associated with all ancestors of the logger are called to dispatch the message (unless the propagate flag for a logger is set to a false value, at which point the passing to ancestor handlers stops).

Just as for loggers, handlers can have levels associated with them. A handler's level acts as a filter in the same way as a logger's level does. If a handler decides to actually dispatch an event, the emit() method is used to send the message to its destination. Most user-defined subclasses of Handler will need to override this emit().

### Custom Levels

Defining your own levels is possible, but should not be necessary, as the existing levels have been chosen on the basis of practical experience. However, if you are convinced that you need custom levels, great care should be exercised when doing this, and it is possibly a very bad idea to define custom levels if you are developing a library. That's because if multiple library authors all define their own custom levels, there is a chance that the logging output from such multiple libraries used together will be difficult for the using developer to control and/or interpret, because a given numeric value might mean different things for different libraries.
Useful Handlers

In addition to the base Handler class, many useful subclasses are provided:

1. **StreamHandler** instances send messages to streams (file-like objects).
2. **FileHandler** instances send messages to disk files.
3. **BaseRotatingHandler** is the base class for handlers that rotate log files at a certain point. It is not meant to be instantiated directly. Instead, use **RotatingFileHandler** or **TimedRotatingFileHandler**.
4. **RotatingFileHandler** instances send messages to disk files, with support for maximum log file sizes and log file rotation.
5. **TimedRotatingFileHandler** instances send messages to disk files, rotating the log file at certain timed intervals.
6. **SocketHandler** instances send messages to TCP/IP sockets.
7. **DatagramHandler** instances send messages to UDP sockets.
8. **SMTPHandler** instances send messages to a designated email address.
9. **SysLogHandler** instances send messages to a Unix syslog daemon, possibly on a remote machine.
10. **NTEventLogHandler** instances send messages to a Windows NT/2000/XP event log.
11. **MemoryHandler** instances send messages to a buffer in memory, which is flushed whenever specific criteria are met.
12. **HTTPHandler** instances send messages to an HTTP server using either GET or POST semantics.
13. **WatchedFileHandler** instances watch the file they are logging to. If the file changes, it is closed and reopened using the file name. This handler is only useful on Unix-like systems; Windows does
not support the underlying mechanism used.

14. **NullHandler** instances do nothing with error messages. They are used by library developers who want to use logging, but want to avoid the ‘No handlers could be found for logger XXX’ message which can be displayed if the library user has not configured logging. See Configuring Logging for a Library for more information.

*New in version 2.7:* The **NullHandler** class.

The **NullHandler**, **StreamHandler** and **FileHandler** classes are defined in the core logging package. The other handlers are defined in a sub-module, **logging.handlers**. (There is also another sub-module, **logging.config**, for configuration functionality.)

Logged messages are formatted for presentation through instances of the **Formatter** class. They are initialized with a format string suitable for use with the % operator and a dictionary.

For formatting multiple messages in a batch, instances of **BufferingFormatter** can be used. In addition to the format string (which is applied to each message in the batch), there is provision for header and trailer format strings.

When filtering based on logger level and/or handler level is not enough, instances of **Filter** can be added to both **Logger** and **Handler** instances (through their **addFilter()** method). Before deciding to process a message further, both loggers and handlers consult all their filters for permission. If any filter returns a false value, the message is not processed further.

The basic **Filter** functionality allows filtering by specific logger name. If this feature is used, messages sent to the named logger and its children are allowed through the filter, and all others dropped.
Exceptions raised during logging

The logging package is designed to swallow exceptions which occur while logging in production. This is so that errors which occur while handling logging events - such as logging misconfiguration, network or other similar errors - do not cause the application using logging to terminate prematurely.

SystemExit and KeyboardInterrupt exceptions are never swallowed. Other exceptions which occur during the emit() method of a Handler subclass are passed to its handleError() method.

The default implementation of handleError() in Handler checks to see if a module-level variable, raiseExceptions, is set. If set, a traceback is printed to sys.stderr. If not set, the exception is swallowed.

**Note:** The default value of raiseExceptions is True. This is because during development, you typically want to be notified of any exceptions that occur. It’s advised that you set raiseExceptions to False for production usage.
Using arbitrary objects as messages

In the preceding sections and examples, it has been assumed that the message passed when logging the event is a string. However, this is not the only possibility. You can pass an arbitrary object as a message, and its `__str__()` method will be called when the logging system needs to convert it to a string representation. In fact, if you want to, you can avoid computing a string representation altogether - for example, the `SocketHandler` emits an event by pickling it and sending it over the wire.
Optimization

Formatting of message arguments is deferred until it cannot be avoided. However, computing the arguments passed to the logging method can also be expensive, and you may want to avoid doing it if the logger will just throw away your event. To decide what to do, you can call the `isEnabledFor()` method which takes a level argument and returns true if the event would be created by the Logger for that level of call. You can write code like this:

```python
if logger.isEnabledFor(logging.DEBUG):
    logger.debug('Message with %s, %s', expensive_func1(), expensive_func2())
```

so that if the logger's threshold is set above `DEBUG`, the calls to `expensive_func1()` and `expensive_func2()` are never made.

**Note:** In some cases, `isEnabledFor()` can itself be more expensive than you’d like (e.g. for deeply nested loggers where an explicit level is only set high up in the logger hierarchy). In such cases (or if you want to avoid calling a method in tight loops), you can cache the result of a call to `isEnabledFor()` in a local or instance variable, and use that instead of calling the method each time. Such a cached value would only need to be recomputed when the logging configuration changes dynamically while the application is running (which is not all that common).

There are other optimizations which can be made for specific applications which need more precise control over what logging information is collected. Here's a list of things you can do to avoid processing during logging which you don’t need:

<table>
<thead>
<tr>
<th>What you don’t want to collect</th>
<th>How to avoid collecting it</th>
</tr>
</thead>
</table>


Information about where calls were made from.

Set `logging._srcfile` to `None`. This avoids calling `sys._getframe()`, which may help to speed up your code in environments like PyPy (which can’t speed up code that uses `sys._getframe()`).

Threading information.

Set `logging.logThreads` to `0`.

Process information.

Set `logging.logProcesses` to `0`.

Also note that the core logging module only includes the basic handlers. If you don’t import `logging.handlers` and `logging.config`, they won’t take up any memory.

See also:

**Module** `logging`

API reference for the logging module.

**Module** `logging.config`

Configuration API for the logging module.

**Module** `logging.handlers`

Useful handlers included with the logging module.

A logging cookbook
Logging Cookbook

Author: Vinay Sajip <vinay_sajip at red-dove dot com>

This page contains a number of recipes related to logging, which have been found useful in the past.
Using logging in multiple modules

Multiple calls to `logging.getLogger('someLogger')` return a reference to the same logger object. This is true not only within the same module, but also across modules as long as it is in the same Python interpreter process. It is true for references to the same object; additionally, application code can define and configure a parent logger in one module and create (but not configure) a child logger in a separate module, and all logger calls to the child will pass up to the parent. Here is a main module:

```python
import logging
import auxiliary_module

# create logger with 'spam_application'
logger = logging.getLogger('spam_application')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
fh.setFormatter(formatter)
ch.setFormatter(formatter)
# add the handlers to the logger
logger.addHandler(fh)
logger.addHandler(ch)

logger.info('creating an instance of auxiliary_module.Auxiliary')
a = auxiliary_module.Auxiliary()
logger.info('created an instance of auxiliary_module.Auxiliary')
logger.info('calling auxiliary_module.Auxiliary.do_something')
a.do_something()
logger.info('finished auxiliary_module.Auxiliary.do_something')
logger.info('calling auxiliary_module.some_function()')
auxiliary_module.some_function()
logger.info('done with auxiliary_module.some_function()')
```
Here is the auxiliary module:

```python
import logging

# create logger
module_logger = logging.getLogger('spam_application.auxiliary')

class Auxiliary:
    def __init__(self):
        self.logger = logging.getLogger('spam_application.auxiliary.Auxiliary')
        self.logger.info('creating an instance of Auxiliary')

    def do_something(self):
        self.logger.info('doing something')
        a = 1 + 1
        self.logger.info('done doing something')

def some_function():
    module_logger.info('received a call to "some_function"')
```

The output looks like this:

```
2005-03-23 23:47:11,671 - spam_application - INFO - calling auxiliary_module.some_function()
2005-03-23 23:47:11,672 - spam_application.auxiliary - INFO - received a call to 'some_function'
2005-03-23 23:47:11,673 - spam_application - INFO - done with auxiliary_module.some_function()
```
Logging from multiple threads

Logging from multiple threads requires no special effort. The following example shows logging from the main (initial) thread and another thread:

```python
import logging
import threading
import time

def worker(arg):
    while not arg['stop']:
        logging.debug('Hi from myfunc')
        time.sleep(0.5)

def main():
    logging.basicConfig(level=logging.DEBUG, format='%({relativeCreated}6d info = {'stop': False}
    thread = threading.Thread(target=worker, args=(info,))
    thread.start()
    while True:
        try:
            logging.debug('Hello from main')
            time.sleep(0.75)
        except KeyboardInterrupt:
            info['stop'] = True
            break
    thread.join()

if __name__ == '__main__':
    main()
```

When run, the script should print something like the following:

```
0 Thread-1 Hi from myfunc
3 MainThread Hello from main
505 Thread-1 Hi from myfunc
755 MainThread Hello from main
1007 Thread-1 Hi from myfunc
1597 MainThread Hello from main
1508 Thread-1 Hi from myfunc
```
This shows the logging output interspersed as one might expect. This approach works for more threads than shown here, of course.
Multiple handlers and formatters

Loggers are plain Python objects. The `addHandler()` method has no minimum or maximum quota for the number of handlers you may add. Sometimes it will be beneficial for an application to log all messages of all severities to a text file while simultaneously logging errors or above to the console. To set this up, simply configure the appropriate handlers. The logging calls in the application code will remain unchanged. Here is a slight modification to the previous simple module-based configuration example:

```python
import logging

logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
ch.setFormatter(formatter)
fh.setFormatter(formatter)
# add the handlers to logger
logger.addHandler(ch)
logger.addHandler(fh)

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

Notice that the ‘application’ code does not care about multiple handlers. All that changed was the addition and configuration of a new handler named `fh`. 
The ability to create new handlers with higher- or lower-severity filters can be very helpful when writing and testing an application. Instead of using many `print` statements for debugging, use `logger.debug`: Unlike the print statements, which you will have to delete or comment out later, the logger.debug statements can remain intact in the source code and remain dormant until you need them again. At that time, the only change that needs to happen is to modify the severity level of the logger and/or handler to debug.
# Logging to multiple destinations

Let’s say you want to log to console and file with different message formats and in differing circumstances. Say you want to log messages with levels of DEBUG and higher to file, and those messages at level INFO and higher to the console. Let’s also assume that the file should contain timestamps, but the console messages should not. Here’s how you can achieve this:

```python
import logging

# set up logging to file - see previous section for more detail
logging.basicConfig(level=logging.DEBUG,
    format='%(asctime)s %(name)-12s %(levelname)-8s
    datefmt='%m-%d %H:%M',
    filename='/temp/myapp.log',
    filemode='w')

# define a Handler which writes INFO messages or higher to the
console = logging.StreamHandler()
console.setLevel(logging.INFO)

# set a format which is simpler for console use
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s

# tell the handler to use this format
console.setFormatter(formatter)

# add the handler to the root logger
logging.getLogger('').addHandler(console)

# Now, we can log to the root logger, or any other logger. First
logging.info('Jackdaws love my big sphinx of quartz.

# Now, define a couple of other loggers which might represent an application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.
logger1.info('How quickly daft jumping zebras vex.
logger2.warning('Jail zesty vixen who grabbed pay from quack.
logger2.error('The five boxing wizards jump quickly.
```
When you run this, on the console you will see

```
root : INFO      Jackdaws love my big sphinx of quartz.
myapp.area1 : INFO   How quickly daft jumping zebras vex.
myapp.area2 : WARNING Jail zesty vixen who grabbed pay from quack.
myapp.area2 : ERROR    The five boxing wizards jump quickly.
```

and in the file you will see something like

```
10-22 22:19 root INFO     Jackdaws love my big sphinx of quartz.
10-22 22:19 myapp.area1 DEBUG Quick zephyrs blow, vexing daft
dafthopping zebras.
10-22 22:19 myapp.area1 INFO How quickly daft jumping zebras vex.
10-22 22:19 myapp.area2 WARNING Jail zesty vixen who grabbed
pay from quack.
10-22 22:19 myapp.area2 ERROR    The five boxing wizards jump
quickly.
```

As you can see, the DEBUG message only shows up in the file. The other messages are sent to both destinations.

This example uses console and file handlers, but you can use any number and combination of handlers you choose.
Here is an example of a module using the logging configuration server:

```python
import logging
import logging.config
import time
import os

# read initial config file
logging.config.fileConfig('logging.conf')

# create and start listener on port 9999
t = logging.config.listen(9999)
t.start()

logger = logging.getLogger('simpleExample')

try:
    # loop through logging calls to see the difference
    # new configurations make, until Ctrl+C is pressed
    while True:
        logger.debug('debug message')
        logger.info('info message')
        logger.warn('warn message')
        logger.error('error message')
        logger.critical('critical message')
        time.sleep(5)
except KeyboardInterrupt:
    # cleanup
    logging.config.stopListening()
t.join()
```

And here is a script that takes a filename and sends that file to the server, properly preceded with the binary-encoded length, as the new logging configuration:

```bash
#!/usr/bin/env python
import socket, sys, struct
```
with open(sys.argv[1], 'rb') as f:
    data_to_send = f.read()

HOST = 'localhost'
PORT = 9999
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print('connecting...')
s.connect((HOST, PORT))
print('sending config...')
s.send(struct.pack('>L', len(data_to_send)))
s.send(data_to_send)
s.close()
print('complete')
Sending and receiving logging events across a network

Let’s say you want to send logging events across a network, and handle them at the receiving end. A simple way of doing this is attaching a `SocketHandler` instance to the root logger at the sending end:

```python
import logging, logging.handlers

rootLogger = logging.getLogger('')
rootLogger.setLevel(logging.DEBUG)
socketHandler = logging.handlers.SocketHandler('localhost',
                                               logging.handlers.DEFAULT_TCP_LOGGING_PORT)
# don't bother with a formatter, since a socket handler sends to
# an unformatted pickle
rootLogger.addHandler(socketHandler)

# Now, we can log to the root logger, or any other logger. First
# logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent a
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

At the receiving end, you can set up a receiver using the `SocketServer` module. Here is a basic working example:

```python
import pickle
import logging
import logging.handlers
import SocketServer
```
import struct

class LogRecordStreamHandler(SocketServer.StreamRequestHandler):
    """Handler for a streaming logging request.

    This basically logs the record using whatever logging policy is configured locally.
    """

def handle(self):
    """Handle multiple requests - each expected to be a 4-byte length,
    followed by the LogRecord in pickle format. Logs the record
    according to whatever policy is configured locally.
    """
    while True:
        chunk = self.connection.recv(4)
        if len(chunk) < 4:
            break
        slen = struct.unpack('>L', chunk)[0]
        chunk = self.connection.recv(slen)
        obj = self.unPickle(chunk)
        record = logging.makeLogRecord(obj)
        self.handleLogRecord(record)

def unPickle(self, data):
    return pickle.loads(data)

def handleLogRecord(self, record):
    # if a name is specified, we use the named logger rather than
    # implied by the record.
    if self.server.logname is not None:
        name = self.server.logname
    else:
        name = record.name
    logger = logging.getLogger(name)
    # N.B. EVERY record gets logged. This is because Logger
    # is normally called AFTER logger-level filtering. If you
    # to do filtering, do it at the client end to save wast
    # cycles and network bandwidth!
    logger.handle(record)

class LogRecordSocketReceiver(SocketServer.ThreadingTCPServer):
    """
Simple TCP socket-based logging receiver suitable for testing

```
allow_reuse_address = 1

def __init__(self, host='localhost',
             port=logging.handlers.DEFAULT_TCP_LOGGING_PORT,
             handler=LogRecordStreamHandler):
    SocketServer.ThreadingTCPServer.__init__(
        self, (host, p
    self.abort = 0
    self.timeout = 1
    self.logname = None

def serve_until_stopped(self):
    import select
    abort = 0
    while not abort:
        rd, wr, ex = select.select([self.socket.fileno()],
                                   [], [],
                                   self.timeout)

        if rd:
            self.handle_request()
        abort = self.abort

def main():
    logging.basicConfig(
        format='%(%relativeCreated)5d %(%name)-15s %(%levelname)-8s
    tcpserver = LogRecordSocketReceiver()
    print('About to start TCP server...')
    tcpserver.serve_until_stopped()

if __name__ == '__main__':
    main()
```

First run the server, and then the client. On the client side, nothing is printed on the console; on the server side, you should see something like:

```
About to start TCP server...
  59  root INFO    Jackdaws love my big sphinx of quartz
  59 myapp.area1 DEBUG Quick zephyrs blow, vexing daft
  69 myapp.area1 INFO    How quickly daft jumping zebras
  69 myapp.area2 WARNING Jail zesty vixen who grabbed pay
  69 myapp.area2 ERROR    The five boxing wizards jump qui
```
Note that there are some security issues with pickle in some scenarios. If these affect you, you can use an alternative serialization scheme by overriding the `makePickle()` method and implementing your alternative there, as well as adapting the above script to use your alternative serialization.
Adding contextual information to your logging output

Sometimes you want logging output to contain contextual information in addition to the parameters passed to the logging call. For example, in a networked application, it may be desirable to log client-specific information in the log (e.g. remote client's username, or IP address). Although you could use the extra parameter to achieve this, it's not always convenient to pass the information in this way. While it might be tempting to create Logger instances on a per-connection basis, this is not a good idea because these instances are not garbage collected. While this is not a problem in practice, when the number of Logger instances is dependent on the level of granularity you want to use in logging an application, it could be hard to manage if the number of Logger instances becomes effectively unbounded.

Using LoggerAdapters to impart contextual information

An easy way in which you can pass contextual information to be output along with logging event information is to use the LoggerAdapter class. This class is designed to look like a Logger, so that you can call debug(), info(), warning(), error(), exception(), critical() and log(). These methods have the same signatures as their counterparts in Logger, so you can use the two types of instances interchangeably.

When you create an instance of LoggerAdapter, you pass it a Logger instance and a dict-like object which contains your contextual information. When you call one of the logging methods on an
instance of `LoggerAdapter`, it delegates the call to the underlying instance of `Logger` passed to its constructor, and arranges to pass the contextual information in the delegated call. Here's a snippet from the code of `LoggerAdapter`:

```python
def debug(self, msg, *args, **kwargs):
    
    Delegate a debug call to the underlying logger, after adding contextual information from this adapter instance.
    
    msg, kwargs = self.process(msg, kwargs)
    self.logger.debug(msg, *args, **kwargs)
```

The `process()` method of `LoggerAdapter` is where the contextual information is added to the logging output. It's passed the message and keyword arguments of the logging call, and it passes back (potentially) modified versions of these to use in the call to the underlying logger. The default implementation of this method leaves the message alone, but inserts an ‘extra’ key in the keyword argument whose value is the dict-like object passed to the constructor. Of course, if you had passed an ‘extra’ keyword argument in the call to the adapter, it will be silently overwritten.

The advantage of using ‘extra’ is that the values in the dict-like object are merged into the `LogRecord` instance's `__dict__`, allowing you to use customized strings with your `Formatter` instances which know about the keys of the dict-like object. If you need a different method, e.g. if you want to prepend or append the contextual information to the message string, you just need to subclass `LoggerAdapter` and override `process()` to do what you need. Here is a simple example:

```python
class CustomAdapter(logging.LoggerAdapter):
    
    This example adapter expects the passed in dict-like object 'connid' key, whose value in brackets is prepended to the log message.
```

def process(self, msg, kwargs):
    return ' [%s] %s' % (self.extra['connid'], msg), kwargs

which you can use like this:

logger = logging.getLogger(__name__)
adapters = CustomAdapter(logger, {'connid': some_conn_id})

Then any events that you log to the adapter will have the value of 
some_conn_id prepended to the log messages.

Using objects other than dicts to pass contextual information

You don’t need to pass an actual dict to a LoggerAdapter - you could 
pass an instance of a class which implements __getitem__ and  
__iter__ so that it looks like a dict to logging. This would be useful if  
you want to generate values dynamically (whereas the values in a  
dict would be constant).

Using Filters to impart contextual information

You can also add contextual information to log output using a user-defined Filter. Filter instances are allowed to modify the LogRecords passed to them, including adding additional attributes which can then be output using a suitable format string, or if needed a custom Formatter.

For example in a web application, the request being processed (or at  
least, the interesting parts of it) can be stored in a threadlocal  
(threading.local) variable, and then accessed from a Filter to add,  
say, information from the request - say, the remote IP address and remote user’s username - to the LogRecord, using the attribute names ‘ip’ and ‘user’ as in the LoggerAdapter example above. In that  
case, the same format string can be used to get similar output to that
shown above. Here’s an example script:

```python
import logging
from random import choice

class ContextFilter(logging.Filter):
    """
    This is a filter which injects contextual information into
    Rather than use actual contextual information, we just use
data in this demo.
    """

    USERS = ['jim', 'fred', 'sheila']
    IPS = ['123.231.231.123', '127.0.0.1', '192.168.0.1']

def filter(self, record):
    record.ip = choice(ContextFilter.IPS)
    record.user = choice(ContextFilter.USERS)
    return True

if __name__ == '__main__':
    levels = (logging.DEBUG, logging.INFO, logging.WARNING, log
    logging.basicConfig(level=logging.DEBUG,
    format='%(asctime)-15s %(name)-5s %(levelname)-8s
    a1 = logging.getLogger('a.b.c')
a2 = logging.getLogger('d.e.f')

    f = ContextFilter()
a1.addFilter(f)
a2.addFilter(f)
a1.debug('A debug message')
a1.info('An info message with %s', 'some parameters')
    for x in range(10):
        lvl = choice(levels)
        lvlname = logging.getLoggerName(lvl)
        a2.log(lvl, 'A message at %s level with %d %s', lvlname,
```

which, when run, produces something like:

```
2010-09-06 22:38:15,292 a.b.c DEBUG      IP: 123.231.231.123 User
2010-09-06 22:38:15,300 a.b.c INFO       IP: 192.168.0.1    User
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1    User
```
2010-09-06 22:38:15,300 d.e.f ERROR    IP: 127.0.0.1    User
2010-09-06 22:38:15,300 d.e.f DEBUG    IP: 127.0.0.1    User
2010-09-06 22:38:15,300 d.e.f ERROR    IP: 123.231.231.123 User
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 192.168.0.1    User
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1    User
2010-09-06 22:38:15,300 d.e.f DEBUG    IP: 192.168.0.1    User
2010-09-06 22:38:15,301 d.e.f ERROR    IP: 127.0.0.1    User
2010-09-06 22:38:15,301 d.e.f DEBUG    IP: 123.231.231.123 User
2010-09-06 22:38:15,301 d.e.f INFO     IP: 123.231.231.123 User
Logging to a single file from multiple processes

Although logging is thread-safe, and logging to a single file from multiple threads in a single process is supported, logging to a single file from *multiple processes* is *not* supported, because there is no standard way to serialize access to a single file across multiple processes in Python. If you need to log to a single file from multiple processes, one way of doing this is to have all the processes log to a `SocketHandler`, and have a separate process which implements a socket server which reads from the socket and logs to file. (If you prefer, you can dedicate one thread in one of the existing processes to perform this function.) This section documents this approach in more detail and includes a working socket receiver which can be used as a starting point for you to adapt in your own applications.

If you are using a recent version of Python which includes the `multiprocessing` module, you could write your own handler which uses the `Lock` class from this module to serialize access to the file from your processes. The existing `FileHandler` and subclasses do not make use of `multiprocessing` at present, though they may do so in the future. Note that at present, the `multiprocessing` module does not provide working lock functionality on all platforms (see https://bugs.python.org/issue3770).
Using file rotation

Sometimes you want to let a log file grow to a certain size, then open a new file and log to that. You may want to keep a certain number of these files, and when that many files have been created, rotate the files so that the number of files and the size of the files both remain bounded. For this usage pattern, the logging package provides a `RotatingFileHandler`:

```python
import glob
import logging
import logging.handlers

LOG_FILENAME = 'logging_rotatingfile_example.out'

# Set up a specific logger with our desired output level
my_logger = logging.getLogger('MyLogger')
my_logger.setLevel(logging.DEBUG)

# Add the log message handler to the logger
handler = logging.handlers.RotatingFileHandler(
    LOG_FILENAME, maxBytes=20, backupCount=5)

my_logger.addHandler(handler)

# Log some messages
for i in range(20):
    my_logger.debug('i = %d' % i)

# See what files are created
logfiles = glob.glob('%s*' % LOG_FILENAME)

for filename in logfiles:
    print(filename)
```

The result should be 6 separate files, each with part of the log history for the application:
The most current file is always `logging_rotatingfile_example.out`, and each time it reaches the size limit it is renamed with the suffix `.1`. Each of the existing backup files is renamed to increment the suffix (.1 becomes .2, etc.) and the .6 file is erased.

Obviously this example sets the log length much too small as an extreme example. You would want to set `maxBytes` to an appropriate value.
An example dictionary-based configuration

Below is an example of a logging configuration dictionary - it’s taken from the documentation on the Django project. This dictionary is passed to `dictConfig()` to put the configuration into effect:

```python
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'verbose': {
            'format': '%(levelname)s %(asctime)s %(module)s %(process)d
        },
        'simple': {
            'format': '%(levelname)s %(message)s'
        },
    },
    'filters': {
        'special': {
            '()': 'project.logging.SpecialFilter',
            'foo': 'bar',
        },
    },
    'handlers': {
        'null': {
            'level': 'DEBUG',
            'class': 'django.utils.log.NullHandler',
        },
        'console':{
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'simple'
        },
        'mail_admins': {  
            'level': 'ERROR',
            'class': 'django.utils.log.AdminEmailHandler',
            'filters': ['special']
        },
    },
    'loggers': {
        'django': {
            'handlers': ['null'],
            'propagate': True,
        };
```
For more information about this configuration, you can see the relevant section of the Django documentation.
Inserting a BOM into messages sent to a SysLogHandler

RFC 5424 requires that a Unicode message be sent to a syslog daemon as a set of bytes which have the following structure: an optional pure-ASCII component, followed by a UTF-8 Byte Order Mark (BOM), followed by Unicode encoded using UTF-8. (See the relevant section of the specification.)

In Python 2.6 and 2.7, code was added to `SysLogHandler` to insert a BOM into the message, but unfortunately, it was implemented incorrectly, with the BOM appearing at the beginning of the message and hence not allowing any pure-ASCII component to appear before it.

As this behaviour is broken, the incorrect BOM insertion code is being removed from Python 2.7.4 and later. However, it is not being replaced, and if you want to produce RFC 5424-compliant messages which include a BOM, an optional pure-ASCII sequence before it and arbitrary Unicode after it, encoded using UTF-8, then you need to do the following:

1. Attach a `Formatter` instance to your `SysLogHandler` instance, with a format string such as:

   ```
   u'ASCII section\u0000Unicode section'
   ```

   The Unicode code point `u'\u0000'`, when encoded using UTF-8, will be encoded as a UTF-8 BOM – the byte-string `\xef\xbb\xbf`.

2. Replace the ASCII section with whatever placeholders you like, but make sure that the data that appears in there after
substitution is always ASCII (that way, it will remain unchanged after UTF-8 encoding).

3. Replace the Unicode section with whatever placeholders you like; if the data which appears there after substitution contains characters outside the ASCII range, that's fine – it will be encoded using UTF-8.

If the formatted message is Unicode, it will be encoded using UTF-8 encoding by `SysLogHandler`. If you follow the above rules, you should be able to produce RFC 5424-compliant messages. If you don't, logging may not complain, but your messages will not be RFC 5424-compliant, and your syslog daemon may complain.
Implementing structured logging

Although most logging messages are intended for reading by humans, and thus not readily machine-parseable, there might be circumstances where you want to output messages in a structured format which is capable of being parsed by a program (without needing complex regular expressions to parse the log message). This is straightforward to achieve using the logging package. There are a number of ways in which this could be achieved, but the following is a simple approach which uses JSON to serialise the event in a machine-parseable manner:

```python
import json
import logging

class StructuredMessage(object):
    def __init__(self, message, **kwargs):
        self.message = message
        self.kwars = kwargs

    def __str__(self):
        return '%s >>> %s' % (self.message, json.dumps(self.kwars))

_logging = StructuredMessage  # optional, to improve readability

logging.basicConfig(level=logging.INFO, format='%(message)s')
logging.info(_('message 1', foo='bar', bar='baz', num=123, fnum=123.456))
```

If the above script is run, it prints:

```
message 1 >>> {"fnum": 123.456, "num": 123, "bar": "baz", "foo"
```

Note that the order of items might be different according to the version of Python used.

If you need more specialised processing, you can use a custom
JSON encoder, as in the following complete example:

```python
from __future__ import unicode_literals

import json
import logging

# This next bit is to ensure the script runs unchanged on 2.x and 3.x
try:
    unicode
except NameError:
    unicode = str

class Encoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, set):
            return tuple(o)
        elif isinstance(o, unicode):
            return o.encode('unicode_escape').decode('ascii')
        return super(Encoder, self).default(o)

class StructuredMessage(object):
    def __init__(self, message, **kwargs):
        self.message = message
        self.kwargs = kwargs

    def __str__(self):
        s = Encoder().encode(self.kwargs)
        return '%s >>> %s' % (self.message, s)

_ = StructuredMessage  # optional, to improve readability

def main():
    logging.basicConfig(level=logging.INFO, format='%(message)s')
    logging.info(_('message 1', set_value=set([1, 2, 3]), snowman)

if __name__ == '__main__':
    main()
```

When the above script is run, it prints:

```
message 1 >>> {"snowman": "\u2603", "set_value": [1, 2, 3]}
```
Note that the order of items might be different according to the version of Python used.
Customizing handlers with \texttt{dictConfig()}

There are times when you want to customize logging handlers in particular ways, and if you use \texttt{dictConfig()} you may be able to do this without subclassing. As an example, consider that you may want to set the ownership of a log file. On POSIX, this is easily done using \texttt{os.chown()}, but the file handlers in the stdlib don’t offer built-in support. You can customize handler creation using a plain function such as:

\begin{verbatim}
def owned_file_handler(filename, mode='a', encoding=None, owner=None):
    if owner:
        import os, pwd, grp
        # convert user and group names to uid and gid
        uid = pwd.getpwnam(owner[0]).pw_uid
        gid = grp.getgrnam(owner[1]).gr_gid
        owner = (uid, gid)
        if not os.path.exists(filename):
            open(filename, 'a').close()
        os.chown(filename, *owner)
    return logging.FileHandler(filename, mode, encoding)
\end{verbatim}

You can then specify, in a logging configuration passed to \texttt{dictConfig()}, that a logging handler be created by calling this function:

\begin{verbatim}
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s %(message)s
        },
    },
    'handlers': {
        'file':{
            # The values below are popped from this dictionary, # used to create the handler, set the handler's level

\end{verbatim}
In this example I am setting the ownership using the `pulse` user and group, just for the purposes of illustration. Putting it together into a working script, `chowntest.py`:

```python
import logging, logging.config, os, shutil

def owned_file_handler(filename, mode='a', encoding=None, owner=None):
    if owner:
        if not os.path.exists(filename):
            open(filename, 'a').close()
            shutil.chown(filename, *owner)
        return logging.FileHandler(filename, mode, encoding)

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s %(message)s
        },
    },
    'handlers': {
        'file':{
            '# The values below are popped from this dictionary and
            # used to create the handler, set the handler's level and
            '# its formatter.
        }
    },
    'root':{
        'handlers': ['file'],
        'level': 'DEBUG',
    },
}
```

In this example I am setting the ownership using the `pulse` user and group, just for the purposes of illustration. Putting it together into a working script, `chowntest.py`:

```python
import logging, logging.config, os, shutil

def owned_file_handler(filename, mode='a', encoding=None, owner=None):
    if owner:
        if not os.path.exists(filename):
            open(filename, 'a').close()
            shutil.chown(filename, *owner)
        return logging.FileHandler(filename, mode, encoding)

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s %(message)s
        },
    },
    'handlers': {
        'file':{
            '# The values below are popped from this dictionary and
            # used to create the handler, set the handler's level and
            '# its formatter.
        }
    },
    'root':{
        'handlers': ['file'],
        'level': 'DEBUG',
    },
}
```
'()': owned_file_handler,
'level': 'DEBUG',
'formatter': 'default',
# The values below are passed to the handler creator callable
# as keyword arguments.
'owner': ['pulse', 'pulse'],
'filename': 'chowntest.log',
'mode': 'w',
'encoding': 'utf-8',
},
},
'root': {
  'handlers': ['file'],
  'level': 'DEBUG',
},
}

logging.config.dictConfig(LOGGING)
logger = logging.getLogger('mylogger')
logger.debug('A debug message')

To run this, you will probably need to run as root:

$ sudo python3.3 chowntest.py
$ cat chowntest.log
2013-11-05 09:34:51,128 DEBUG mylogger A debug message
$ ls -l chowntest.log
-rw-r--r-- 1 pulse pulse 55 2013-11-05 09:34 chowntest.log

Note that this example uses Python 3.3 because that's where shutil.chown() makes an appearance. This approach should work with any Python version that supports dictConfig() - namely, Python 2.7, 3.2 or later. With pre-3.3 versions, you would need to implement the actual ownership change using e.g. os.chown().

In practice, the handler-creating function may be in a utility module somewhere in your project. Instead of the line in the configuration:

'()': owned_file_handler,
you could use e.g.:

```python
'()': 'ext://project.util.owned_file_handler',
```

where `project.util` can be replaced with the actual name of the package where the function resides. In the above working script, using `ext://__main__.owned_file_handler` should work. Here, the actual callable is resolved by `dictConfig()` from the `ext://` specification.

This example hopefully also points the way to how you could implement other types of file change - e.g. setting specific POSIX permission bits - in the same way, using `os.chmod()`.

Of course, the approach could also be extended to types of handler other than a `FileHandler` - for example, one of the rotating file handlers, or a different type of handler altogether.
You can configure filters using `dictConfig()`, though it might not be obvious at first glance how to do it (hence this recipe). Since `Filter` is the only filter class included in the standard library, and it is unlikely to cater to many requirements (it’s only there as a base class), you will typically need to define your own `Filter` subclass with an overridden `filter()` method. To do this, specify the `()` key in the configuration dictionary for the filter, specifying a callable which will be used to create the filter (a class is the most obvious, but you can provide any callable which returns a `Filter` instance). Here is a complete example:

```python
import logging
import logging.config
import sys

class MyFilter(logging.Filter):
    def __init__(self, param=None):
        self.param = param

    def filter(self, record):
        if self.param is None:
            allow = True
        else:
            allow = self.param not in record.msg
        if allow:
            record.msg = 'changed: ' + record.msg
        return allow

LOGGING = {
    'version': 1,
    'filters': {
        'myfilter': {
            '()': MyFilter,
            'param': 'noshow',
        }
    },
    'handlers': {
        'console': {
```
{'class': 'logging.StreamHandler',
'filters': ['myfilter']
},
},
'root': {
    'level': 'DEBUG',
    'handlers': ['console']
},
}

if __name__ == '__main__':
    logging.config.dictConfig(LOGGING)
    logging.debug('hello')
    logging.debug('hello - noshow')

This example shows how you can pass configuration data to the callable which constructs the instance, in the form of keyword parameters. When run, the above script will print:

changed: hello

which shows that the filter is working as configured.

A couple of extra points to note:

- If you can't refer to the callable directly in the configuration (e.g. if it lives in a different module, and you can't import it directly where the configuration dictionary is), you can use the form `ext://...` as described in Access to external objects. For example, you could have used the text `ext://__main__.MyFilter` instead of `MyFilter` in the above example.
- As well as for filters, this technique can also be used to configure custom handlers and formatters. See User-defined objects for more information on how logging supports using user-defined objects in its configuration, and see the other cookbook recipe Customizing handlers with dictConfig() above.
Customized exception formatting

There might be times when you want to do customized exception formatting - for argument’s sake, let’s say you want exactly one line per logged event, even when exception information is present. You can do this with a custom formatter class, as shown in the following example:

```python
import logging

class OneLineExceptionFormatter(logging.Formatter):
    def formatException(self, exc_info):
        """
        Format an exception so that it prints on a single line.
        """
        result = super(OneLineExceptionFormatter, self).formatException(exc_info)
        return repr(result)  # or format into one line however you
        def format(self, self, record):
            s = super(OneLineExceptionFormatter, self).format(record)
            if record.exc_text:
                s = s.replace('
', '') + '|
            return s

def configure_logging():
    fh = logging.FileHandler('output.txt', 'w')
    f = OneLineExceptionFormatter('%(asctime)s|%(levelname)s|%(message)s',
                                  '%d/%m/%Y %H:%M:%S')
    fh.setFormatter(f)
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)
    root.addHandler(fh)

def main():
    configure_logging()
    logging.info('Sample message')
    try:
        x = 1 / 0
    except ZeroDivisionError as e:
        logging.exception('ZeroDivisionError: %s', e)

if __name__ == '__main__':
```
When run, this produces a file with exactly two lines:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Level</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>28/01/2015 07:21:23</td>
<td>INFO</td>
<td></td>
<td>Sample message</td>
</tr>
<tr>
<td>28/01/2015 07:21:23</td>
<td>ERROR</td>
<td></td>
<td>ZeroDivisionError: integer division o</td>
</tr>
</tbody>
</table>

While the above treatment is simplistic, it points the way to how exception information can be formatted to your liking. The `traceback` module may be helpful for more specialized needs.
Speaking logging messages

There might be situations when it is desirable to have logging messages rendered in an audible rather than a visible format. This is easy to do if you have text- to-speech (TTS) functionality available in your system, even if it doesn't have a Python binding. Most TTS systems have a command line program you can run, and this can be invoked from a handler using subprocess. It's assumed here that TTS command line programs won't expect to interact with users or take a long time to complete, and that the frequency of logged messages will be not so high as to swamp the user with messages, and that it's acceptable to have the messages spoken one at a time rather than concurrently. The example implementation below waits for one message to be spoken before the next is processed, and this might cause other handlers to be kept waiting. Here is a short example showing the approach, which assumes that the espeak TTS package is available:

```python
import logging
import subprocess
import sys

class TTSHandler(logging.Handler):
    def emit(self, record):
        msg = self.format(record)
        # Speak slowly in a female English voice
        cmd = ['espeak', '-s150', '-ven+f3', msg]
        p = subprocess.Popen(cmd, stdout=subprocess.PIPE,
                             stderr=subprocess.STDOUT)
        # wait for the program to finish
        p.communicate()

def configure_logging():
    h = TTSHandler()
    root = logging.getLogger()
    root.addHandler(h)
    # the default formatter just returns the message
    root.setLevel(logging.DEBUG)
```
def main():
    logging.info('Hello')
    logging.debug('Goodbye')

if __name__ == '__main__':
    configure_logging()
    sys.exit(main())

When run, this script should say “Hello” and then “Goodbye” in a female voice.

The above approach can, of course, be adapted to other TTS systems and even other systems altogether which can process messages via external programs run from a command line.
Buffering logging messages and outputting them conditionally

There might be situations where you want to log messages in a temporary area and only output them if a certain condition occurs. For example, you may want to start logging debug events in a function, and if the function completes without errors, you don’t want to clutter the log with the collected debug information, but if there is an error, you want all the debug information to be output as well as the error.

Here is an example which shows how you could do this using a decorator for your functions where you want logging to behave this way. It makes use of the `logging.handlers.MemoryHandler`, which allows buffering of logged events until some condition occurs, at which point the buffered events are flushed - passed to another handler (the target handler) for processing. By default, the `MemoryHandler` flushed when its buffer gets filled up or an event whose level is greater than or equal to a specified threshold is seen. You can use this recipe with a more specialised subclass of `MemoryHandler` if you want custom flushing behavior.

The example script has a simple function, `foo`, which just cycles through all the logging levels, writing to `sys.stderr` to say what level it’s about to log at, and then actually logging a message at that level. You can pass a parameter to `foo` which, if true, will log at ERROR and CRITICAL levels - otherwise, it only logs at DEBUG, INFO and WARNING levels.

The script just arranges to decorate `foo` with a decorator which will do the conditional logging that’s required. The decorator takes a logger as a parameter and attaches a memory handler for the
duration of the call to the decorated function. The decorator can be additionally parameterised using a target handler, a level at which flushing should occur, and a capacity for the buffer. These default to a `StreamHandler` which writes to `sys.stderr`, `logging.ERROR` and 100 respectively.

Here's the script:

```python
import logging
from logging.handlers import MemoryHandler
import sys

logger = logging.getLogger(__name__)
logger.addHandler(logging.NullHandler())

def log_if_errors(logger, target_handler=None, flush_level=None):
    if target_handler is None:
        target_handler = logging.StreamHandler()
    if flush_level is None:
        flush_level = logging.ERROR
    if capacity is None:
        capacity = 100
    handler = MemoryHandler(capacity, flushLevel=flush_level, t

def decorator(fn):
    def wrapper(*args, **kwargs):
        logger.addHandler(handler)
        try:
            return fn(*args, **kwargs)
        except Exception:
            logger.exception('call failed')
        raise
    finally:
        super(MemoryHandler, handler).flush()
        logger.removeHandler(handler)
    return wrapper

def write_line(s):
    sys.stderr.write('%s
' % s)

def foo(fail=False):
    write_line('about to log at DEBUG ...')
```
logger.debug('Actually logged at DEBUG')
write_line('about to log at INFO ...')
logger.info('Actually logged at INFO')
write_line('about to log at WARNING ...')
logger.warning('Actually logged at WARNING')

if fail:
    write_line('about to log at ERROR ...')
    logger.error('Actually logged at ERROR')
    write_line('about to log at CRITICAL ...')
    logger.critical('Actually logged at CRITICAL')

return fail

decorated_foo = log_if_errors(logger)(foo)

if __name__ == '__main__':
    logger.setLevel(logging.DEBUG)
    write_line('Calling undecorated foo with False')
    assert not foo(False)
    write_line('Calling undecorated foo with True')
    assert foo(True)
    write_line('Calling decorated foo with False')
    assert not decorated_foo(False)
    write_line('Calling decorated foo with True')
    assert decorated_foo(True)

When this script is run, the following output should be observed:

Calling undecorated foo with False
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
Calling undecorated foo with True
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
about to log at ERROR ...
about to log at CRITICAL ...
Calling decorated foo with False
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
Calling decorated foo with True
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
About to log at ERROR ... Actually logged at DEBUG
Actually logged at INFO
Actually logged at WARNING
Actually logged at ERROR
About to log at CRITICAL ...
Actually logged at CRITICAL

As you can see, actual logging output only occurs when an event is logged whose severity is ERROR or greater, but in that case, any previous events at lower severities are also logged.

You can of course use the conventional means of decoration:

```python
@log_if_errors(logger)
def foo(fail=False):
    ...
```
Sometimes you want to format times using UTC, which can be done using a class such as `UTCFormatter`, shown below:

```python
import logging
import time
class UTCFormatter(logging.Formatter):
    converter = time.gmtime
```

and you can then use the `UTCFormatter` in your code instead of `Formatter`. If you want to do that via configuration, you can use the `dictConfig()` API with an approach illustrated by the following complete example:

```python
import logging
import logging.config
import time
class UTCFormatter(logging.Formatter):
    converter = time.gmtime

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'utc': {
            '()': UTCFormatter,
            'format': '%(asctime)s %(message)s',
        },
        'local': {
            'format': '%(asctime)s %(message)s',
        },
    },
    'handlers': {
        'console1': {
            'class': 'logging.StreamHandler',
            'formatter': 'utc',
        },
    },
}```
{ },
'console2': {
    'class': 'logging.StreamHandler',
    'formatter': 'local',
},
},
'root': {
    'handlers': ['console1', 'console2'],
}
}

if __name__ == '__main__':
    logging.config.dictConfig(LOGGING)
    logging.warning('The local time is %s', time.asctime())

When this script is run, it should print something like:

```
2015-10-17 12:53:29,501 The local time is Sat Oct 17 13:53:29 2
```

showing how the time is formatted both as local time and UTC, one for each handler.
Using a context manager for selective logging

There are times when it would be useful to temporarily change the logging configuration and revert it back after doing something. For this, a context manager is the most obvious way of saving and restoring the logging context. Here is a simple example of such a context manager, which allows you to optionally change the logging level and add a logging handler purely in the scope of the context manager:

```python
import logging
import sys

class LoggingContext(object):
    def __init__(self, logger, level=None, handler=None, close=True):
        self.logger = logger
        self.level = level
        self.handler = handler
        self.close = close

    def __enter__(self):
        if self.level is not None:
            self.old_level = self.logger.level
            self.logger.setLevel(self.level)
        if self.handler:
            self.logger.addHandler(self.handler)

    def __exit__(self, et, ev, tb):
        if self.level is not None:
            self.logger.setLevel(self.old_level)
        if self.handler:
            self.logger.removeHandler(self.handler)
        if self.handler and self.close:
            self.handler.close()

# implicit return of None => don't swallow exceptions
```

If you specify a level value, the logger’s level is set to that value in
the scope of the with block covered by the context manager. If you specify a handler, it is added to the logger on entry to the block and removed on exit from the block. You can also ask the manager to close the handler for you on block exit - you could do this if you don’t need the handler any more.

To illustrate how it works, we can add the following block of code to the above:

```python
if __name__ == '__main__':
    logger = logging.getLogger('foo')
    logger.addHandler(logging.StreamHandler())
    logger.setLevel(logging.INFO)
    logger.info('1. This should appear just once on stderr.')
    logger.debug('2. This should not appear.')
    with LoggingContext(logger, level=logging.DEBUG):
        logger.debug('3. This should appear once on stderr.')
    logger.debug('4. This should not appear.')
    h = logging.StreamHandler(sys.stdout)
    with LoggingContext(logger, level=logging.DEBUG, handler=h, close=True):
        logger.debug('5. This should appear twice - once on stdout and once on stderr.')
    logger.info('6. This should appear just once on stderr.')
    logger.debug('7. This should not appear.')
```

We initially set the logger’s level to INFO, so message #1 appears and message #2 doesn’t. We then change the level to DEBUG temporarily in the following with block, and so message #3 appears. After the block exits, the logger’s level is restored to INFO and so message #4 doesn’t appear. In the next with block, we set the level to DEBUG again but also add a handler writing to sys.stdout. Thus, message #5 appears twice on the console (once via stderr and once via stdout). After the with statement’s completion, the status is as it was before so message #6 appears (like message #1) whereas message #7 doesn’t (just like message #2).

If we run the resulting script, the result is as follows:
1. This should appear just once on stderr.
2. This should appear once on stderr.
3. This should appear twice - once on stderr and once on stdout.
4. This should appear twice - once on stderr and once on stdout.
5. This should appear just once on stderr.

If we run it again, but pipe stderr to /dev/null, we see the following, which is the only message written to stdout:

$ python logctx.py
5. This should appear twice - once on stderr and once on stdout.

Once again, but piping stdout to /dev/null, we get:

$ python logctx.py
1. This should appear just once on stderr.
2. This should appear once on stderr.
3. This should appear twice - once on stderr and once on stdout.
4. This should appear twice - once on stderr and once on stdout.

In this case, the message #5 printed to stdout doesn’t appear, as expected.

Of course, the approach described here can be generalised, for example to attach logging filters temporarily. Note that the above code works in Python 2 as well as Python 3.
Regular Expression HOWTO

Author: A.M. Kuchling <amk@amk.ca>

Abstract

This document is an introductory tutorial to using regular expressions in Python with the `re` module. It provides a gentler introduction than the corresponding section in the Library Reference.
Introduction

The `re` module was added in Python 1.5, and provides Perl-style regular expression patterns. Earlier versions of Python came with the `regex` module, which provided Emacs-style patterns. The `regex` module was removed completely in Python 2.5.

Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the `re` module. Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything you like. You can then ask questions such as “Does this string match the pattern?”, or “Is there a match for the pattern anywhere in this string?”. You can also use REs to modify a string or to split it apart in various ways.

Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C. For advanced use, it may be necessary to pay careful attention to how the engine will execute a given RE, and write the RE in a certain way in order to produce bytecode that runs faster. Optimization isn’t covered in this document, because it requires that you have a good understanding of the matching engine’s internals.

The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are also tasks that can be done with regular expressions, but the expressions turn out to be very complicated. In these cases, you may be better off writing Python code to do the processing; while Python code will be slower than an elaborate regular expression, it will also probably be more understandable.
Simple Patterns

We'll start by learning about the simplest possible regular expressions. Since regular expressions are used to operate on strings, we'll begin with the most common task: matching characters.

For a detailed explanation of the computer science underlying regular expressions (deterministic and non-deterministic finite automata), you can refer to almost any textbook on writing compilers.

Matching Characters

Most letters and characters will simply match themselves. For example, the regular expression \texttt{test} will match the string \texttt{test} exactly. (You can enable a case-insensitive mode that would let this RE match \texttt{Test} or \texttt{TEST} as well; more about this later.)

There are exceptions to this rule; some characters are special \textit{metacharacters}, and don’t match themselves. Instead, they signal that some out-of-the-ordinary thing should be matched, or they affect other portions of the RE by repeating them or changing their meaning. Much of this document is devoted to discussing various metacharacters and what they do.

Here's a complete list of the metacharacters; their meanings will be discussed in the rest of this HOWTO.

\begin{verbatim}
 .  ^  $  *  +  ?  {  }  [ ]  \  |  ( )
\end{verbatim}

The first metacharacters we'll look at are [ and ]. They're used for specifying a character class, which is a set of characters that you wish to match. Characters can be listed individually, or a range of
characters can be indicated by giving two characters and separating them by a '-' . For example, [abc] will match any of the characters a, b, or c; this is the same as [a-c] , which uses a range to express the same set of characters. If you wanted to match only lowercase letters, your RE would be [a-z] .

Metacharacters are not active inside classes. For example, [akm$] will match any of the characters 'a', 'k', 'm', or '$'; '$' is usually a metacharacter, but inside a character class it's stripped of its special nature.

You can match the characters not listed within the class by complementing the set. This is indicated by including a '^' as the first character of the class; '^' outside a character class will simply match the '^' character. For example, [^5] will match any character except '5'.

Perhaps the most important metacharacter is the backslash, \\ . As in Python string literals, the backslash can be followed by various characters to signal various special sequences. It's also used to escape all the metacharacters so you can still match them in patterns; for example, if you need to match a [ or \, you can precede them with a backslash to remove their special meaning: \[ or \\.

Some of the special sequences beginning with '\\' represent predefined sets of characters that are often useful, such as the set of digits, the set of letters, or the set of anything that isn't whitespace. The following predefined special sequences are a subset of those available. The equivalent classes are for byte string patterns. For a complete list of sequences and expanded class definitions for Unicode string patterns, see the last part of Regular Expression Syntax.
\d
Matches any decimal digit; this is equivalent to the class \[^0-9\].

\D
Matches any non-digit character; this is equivalent to the class \[^\^0-9\].

\s
Matches any whitespace character; this is equivalent to the class \[^\t\n\r\f\v\].

\S
Matches any non-whitespace character; this is equivalent to the class \[^\^\t\n\r\f\v\].

\w
Matches any alphanumeric character; this is equivalent to the class \[a-zA-Z0-9_\].

\W
Matches any non-alphanumeric character; this is equivalent to the class \[^a-zA-Z0-9_\].

These sequences can be included inside a character class. For example, \[s, .\] is a character class that will match any whitespace character, or ' , ' or '. '.

The final metacharacter in this section is \. It matches anything except a newline character, and there's an alternate mode (re.DOTALL) where it will match even a newline. \. is often used where you want to match “any character”.

**Repeating Things**

Being able to match varying sets of characters is the first thing regular expressions can do that isn’t already possible with the methods available on strings. However, if that was the only additional capability of regexes, they wouldn’t be much of an advance. Another
capability is that you can specify that portions of the RE must be repeated a certain number of times.

The first metacharacter for repeating things that we'll look at is \*\. \* doesn't match the literal character \*\; instead, it specifies that the previous character can be matched zero or more times, instead of exactly once.

For example, `ca*t` will match `ct` (0 a characters), `ca` (1 a), `caaat` (3 a characters), and so forth. The RE engine has various internal limitations stemming from the size of C's `int` type that will prevent it from matching over 2 billion a characters; you probably don't have enough memory to construct a string that large, so you shouldn't run into that limit.

Repetitions such as \*\ are greedy; when repeating a RE, the matching engine will try to repeat it as many times as possible. If later portions of the pattern don't match, the matching engine will then back up and try again with fewer repetitions.

A step-by-step example will make this more obvious. Let's consider the expression `a[bcd]*b`. This matches the letter 'a', zero or more letters from the class `[bcd]`, and finally ends with a 'b'. Now imagine matching this RE against the string `abcbd`.

<table>
<thead>
<tr>
<th>Step</th>
<th>Matched</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>The a in the RE matches.</td>
</tr>
<tr>
<td>2</td>
<td>abcbd</td>
<td>The engine matches <code>[bcd]*</code>, going as far as it can, which is to the end of the string.</td>
</tr>
<tr>
<td>3</td>
<td><strong>Failure</strong></td>
<td>The engine tries to match b, but the current position is at the end of the string, so it fails.</td>
</tr>
<tr>
<td>4</td>
<td>abcba</td>
<td>Back up, so that <code>[bcd]*</code> matches one less character.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Try b again, but the current position is at the</td>
</tr>
</tbody>
</table>
Failure last character, which is a 'd'.

Back up again, so that [bcd]* is only matching bc.

Try b again. This time the character at the current position is 'b', so it succeeds.

The end of the RE has now been reached, and it has matched abcb. This demonstrates how the matching engine goes as far as it can at first, and if no match is found it will then progressively back up and retry the rest of the RE again and again. It will back up until it has tried zero matches for [bcd]*, and if that subsequently fails, the engine will conclude that the string doesn’t match the RE at all.

Another repeating metacharacter is +, which matches one or more times. Pay careful attention to the difference between * and +; * matches zero or more times, so whatever’s being repeated may not be present at all, while + requires at least one occurrence. To use a similar example, ca+t will match cat (1 a), caaat (3 a’s), but won’t match ct.

There are two more repeating qualifiers. The question mark character, ?, matches either once or zero times; you can think of it as marking something as being optional. For example, home-?brew matches either homebrew or home-brew.

The most complicated repeated qualifier is \{m,n\}, where m and n are decimal integers. This qualifier means there must be at least m repetitions, and at most n. For example, a/{1,3}b will match a/b, a//b, and a///b. It won’t match ab, which has no slashes, or a////b, which has four.

You can omit either m or n; in that case, a reasonable value is assumed for the missing value. Omitting m is interpreted as a lower
limit of 0, while omitting $n$ results in an upper bound of infinity — actually, the upper bound is the 2-billion limit mentioned earlier, but that might as well be infinity.

Readers of a reductionist bent may notice that the three other qualifiers can all be expressed using this notation. $\{0,\}$ is the same as $\ast$, $\{1,\}$ is equivalent to $+$, and $\{0,1\}$ is the same as $\?$. It’s better to use $\ast$, $+$, or $\?$ when you can, simply because they’re shorter and easier to read.
Using Regular Expressions

Now that we've looked at some simple regular expressions, how do we actually use them in Python? The `re` module provides an interface to the regular expression engine, allowing you to compile REs into objects and then perform matches with them.

Compiling Regular Expressions

Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

```python
>>> import re
>>> p = re.compile('ab*')
>>> p
<_sre.SRE_Pattern object at 0x...>
```

`re.compile()` also accepts an optional `flags` argument, used to enable various special features and syntax variations. We'll go over the available settings later, but for now a single example will do:

```python
>>> p = re.compile('ab*', re.IGNORECASE)
```

The RE is passed to `re.compile()` as a string. REs are handled as strings because regular expressions aren't part of the core Python language, and no special syntax was created for expressing them. (There are applications that don’t need REs at all, so there’s no need to bloat the language specification by including them.) Instead, the `re` module is simply a C extension module included with Python, just like the `socket` or `zlib` modules.

Putting REs in strings keeps the Python language simpler, but has one disadvantage which is the topic of the next section.
As stated earlier, regular expressions use the backslash character (\") to indicate special forms or to allow special characters to be used without invoking their special meaning. This conflicts with Python's usage of the same character for the same purpose in string literals.

Let's say you want to write a RE that matches the string \section, which might be found in a LaTeX file. To figure out what to write in the program code, start with the desired string to be matched. Next, you must escape any backslashes and other metacharacters by preceding them with a backslash, resulting in the string \section. The resulting string that must be passed to `re.compile()` must be \section. However, to express this as a Python string literal, both backslashes must be escaped again.

<table>
<thead>
<tr>
<th>Characters</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>\section</td>
<td>Text string to be matched</td>
</tr>
<tr>
<td>\section</td>
<td>Escaped backslash for <code>re.compile()</code></td>
</tr>
<tr>
<td>&quot;\section&quot;</td>
<td>Escaped backslashes for a string literal</td>
</tr>
</tbody>
</table>

In short, to match a literal backslash, one has to write '\\' as the RE string, because the regular expression must be \, and each backslash must be expressed as \ inside a regular Python string literal. In REs that feature backslashes repeatedly, this leads to lots of repeated backslashes and makes the resulting strings difficult to understand.

The solution is to use Python's raw string notation for regular expressions; backslashes are not handled in any special way in a string literal prefixed with 'r', so r"\n" is a two-character string.
containing \"\" and \"n\", while \"\n\" is a one-character string containing a newline. Regular expressions will often be written in Python code using this raw string notation.

<table>
<thead>
<tr>
<th>Regular String</th>
<th>Raw string</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ab*&quot;</td>
<td>r&quot;ab*&quot;</td>
</tr>
<tr>
<td>&quot;\section&quot;</td>
<td>r&quot;\section&quot;</td>
</tr>
<tr>
<td>&quot;\w+\s+\1&quot;</td>
<td>r&quot;\w+\s+\1&quot;</td>
</tr>
</tbody>
</table>

Performing Matches

Once you have an object representing a compiled regular expression, what do you do with it? Pattern objects have several methods and attributes. Only the most significant ones will be covered here; consult the re docs for a complete listing.

<table>
<thead>
<tr>
<th>Method/Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>match()</td>
<td>Determine if the RE matches at the beginning of the string.</td>
</tr>
<tr>
<td>search()</td>
<td>Scan through a string, looking for any location where this RE matches.</td>
</tr>
<tr>
<td>findall()</td>
<td>Find all substrings where the RE matches, and returns them as a list.</td>
</tr>
<tr>
<td>finditer()</td>
<td>Find all substrings where the RE matches, and returns them as an iterator.</td>
</tr>
</tbody>
</table>

match() and search() return None if no match can be found. If they’re successful, a match object instance is returned, containing information about the match: where it starts and ends, the substring it matched, and more.

You can learn about this by interactively experimenting with the re module. If you have Tkinter available, you may also want to look at Tools/scripts/redemo.py, a demonstration program included with the Python distribution. It allows you to enter REs and strings, and
displays whether the RE matches or fails. redemo.py can be quite useful when trying to debug a complicated RE. Phil Schwartz's Kodos is also an interactive tool for developing and testing RE patterns.

This HOWTO uses the standard Python interpreter for its examples. First, run the Python interpreter, import the re module, and compile a RE:

```python
Python 2.2.2 (#1, Feb 10 2003, 12:57:01)
>>> import re
>>> p = re.compile('[a-z]+')
>>> p #doctest: +ELLIPSIS
<_sre.SRE_Pattern object at 0x...>
```

Now, you can try matching various strings against the RE [a-z]+. An empty string shouldn't match at all, since + means 'one or more repetitions'. match() should return None in this case, which will cause the interpreter to print no output. You can explicitly print the result of match() to make this clear.

```python
>>> p.match('')
>>> print p.match('')
None
```

Now, let's try it on a string that it should match, such as tempo. In this case, match() will return a match object, so you should store the result in a variable for later use.

```python
>>> m = p.match('tempo')
>>> m
<_sre.SRE_Match object at 0x...>
```

Now you can query the match object for information about the matching string. match object instances also have several methods and attributes; the most important ones are:
<table>
<thead>
<tr>
<th>Method/Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>group()</td>
<td>Return the string matched by the RE</td>
</tr>
<tr>
<td>start()</td>
<td>Return the starting position of the match</td>
</tr>
<tr>
<td>end()</td>
<td>Return the ending position of the match</td>
</tr>
<tr>
<td>span()</td>
<td>Return a tuple containing the (start, end) positions of the match</td>
</tr>
</tbody>
</table>

Trying these methods will soon clarify their meaning:

```python
>>> m.group()
'tempo'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)
```

`group()` returns the substring that was matched by the RE. `start()` and `end()` return the starting and ending index of the match. `span()` returns both start and end indexes in a single tuple. Since the `match()` method only checks if the RE matches at the start of a string, `start()` will always be zero. However, the `search()` method of patterns scans through the string, so the match may not start at zero in that case.

```python
>>> print p.match('::: message')
None
>>> m = p.search('::: message'); print m
<_sre.SRE_Match object at 0x...>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

In actual programs, the most common style is to store the `match object` in a variable, and then check if it was `None`. This usually looks like:

```python
p = re.compile( ... )
```
m = p.match( 'string goes here' )
if m:
    print 'Match found: ', m.group()
else:
    print 'No match'

Two pattern methods return all of the matches for a pattern. findall() returns a list of matching strings:

```python
>>> p = re.compile('\d+')
>>> p.findall('12 drummers drumming, 11 pipers piping, 10 lords ['12', '11', '10']
```

findall() has to create the entire list before it can be returned as the result. The finditer() method returns a sequence of match object instances as an iterator. [1]

```python
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10 ...')
>>> iterator
<callable-iterator object at 0x...>
>>> for match in iterator:
...    print match.span()
...
(0, 2)
(22, 24)
(29, 31)
```

Module-Level Functions

You don’t have to create a pattern object and call its methods; the re module also provides top-level functions called match(), search(), findall(), sub(), and so forth. These functions take the same arguments as the corresponding pattern method, with the RE string added as the first argument, and still return either None or a match object instance.

```python
>>> print re.match(r'From\s+', 'Fromage amk')
```
Under the hood, these functions simply create a pattern object for you and call the appropriate method on it. They also store the compiled object in a cache, so future calls using the same RE are faster.

Should you use these module-level functions, or should you get the pattern and call its methods yourself? That choice depends on how frequently the RE will be used, and on your personal coding style. If the RE is being used at only one point in the code, then the module functions are probably more convenient. If a program contains a lot of regular expressions, or re-uses the same ones in several locations, then it might be worthwhile to collect all the definitions in one place, in a section of code that compiles all the REs ahead of time. To take an example from the standard library, here’s an extract from the deprecated xmllib module:

```python
ref = re.compile( ... )
entityref = re.compile( ... )
charref = re.compile( ... )
starttagopen = re.compile( ... )
```

I generally prefer to work with the compiled object, even for one-time uses, but few people will be as much of a purist about this as I am.

## Compilation Flags

Compilation flags let you modify some aspects of how regular expressions work. Flags are available in the `re` module under two names, a long name such as `IGNORECASE` and a short, one-letter form such as `i`. (If you're familiar with Perl's pattern modifiers, the one-letter forms use the same letters; the short form of `re.VERBOSE` is
Multiple flags can be specified by bitwise OR-ing them; \texttt{re.I | re.M} sets both the \texttt{I} and \texttt{M} flags, for example.

Here's a table of the available flags, followed by a more detailed explanation of each one.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOTALL, S</td>
<td>Make . match any character, including newlines</td>
</tr>
<tr>
<td>IGNORECASE, I</td>
<td>Do case-insensitive matches</td>
</tr>
<tr>
<td>LOCALE, L</td>
<td>Do a locale-aware match</td>
</tr>
<tr>
<td>MULTILINE, M</td>
<td>Multi-line matching, affecting ^ and $</td>
</tr>
<tr>
<td>VERBOSE, X</td>
<td>Enable verbose REs, which can be organized more cleanly and understandably.</td>
</tr>
<tr>
<td>UNICODE, U</td>
<td>Makes several escapes like \w, \b, \s and \d dependent on the Unicode character database.</td>
</tr>
</tbody>
</table>

**I**

**IGNORECASE**

Perform case-insensitive matching; character class and literal strings will match letters by ignoring case. For example, \[A-Z\] will match lowercase letters, too, and \texttt{Spam} will match \texttt{Spam}, \texttt{spam}, or \texttt{spAM}. This lowercasing doesn't take the current locale into account; it will if you also set the \texttt{LOCALE} flag.

**L**

**LOCALE**

Make \w, \w, \b, and \b, dependent on the current locale.

Locales are a feature of the C library intended to help in writing programs that take account of language differences. For example, if you're processing French text, you'd want to be able to write \texttt{\w+} to match words, but \texttt{\w} only matches the character
class [A-Za-z]; it won’t match ‘é’ or ‘ç’. If your system is configured properly and a French locale is selected, certain C functions will tell the program that ‘é’ should also be considered a letter. Setting the **locale** flag when compiling a regular expression will cause the resulting compiled object to use these C functions for \w; this is slower, but also enables \w+ to match French words as you’d expect.

**M**
**MULTILINE**
(\^ and \$ haven’t been explained yet; they’ll be introduced in section **More Metacharacters**.)

Usually \^ matches only at the beginning of the string, and \$ matches only at the end of the string and immediately before the newline (if any) at the end of the string. When this flag is specified, \^ matches at the beginning of the string and at the beginning of each line within the string, immediately following each newline. Similarly, the \$ metacharacter matches either at the end of the string and at the end of each line (immediately preceding each newline).

**S**
**DOTALL**
Makes the ‘.’ special character match any character at all, including a newline; without this flag, ‘.’ will match anything except a newline.

**U**
**UNICODE**
Make \w, \W, \b, \B, \d, \D, \s and \S dependent on the Unicode character properties database.

**X**
**VERBOSE**
This flag allows you to write regular expressions that are more readable by granting you more flexibility in how you can format them. When this flag has been specified, whitespace within the RE string is ignored, except when the whitespace is in a character class or preceded by an unescaped backslash; this lets you organize and indent the RE more clearly. This flag also lets you put comments within a RE that will be ignored by the engine; comments are marked by a '#' that's neither in a character class or preceded by an unescaped backslash.

For example, here's a RE that uses `re.VERBOSE`; see how much easier it is to read?

```python
charref = re.compile(r'""'
    &[#]
    (  
        0[0-7]+     # Octal form
        | [0-9]+     # Decimal form
        | x[0-9a-fA-F]+ # Hexadecimal form
    )
    ;
""', re.VERBOSE)
```

Without the verbose setting, the RE would look like this:

```python
charref = re.compile("&#(0[0-7]+"  
    
    
    "|([0-9]+"  
    "|x([0-9a-fA-F]+);")
```

In the above example, Python’s automatic concatenation of string literals has been used to break up the RE into smaller pieces, but it’s still more difficult to understand than the version using `re.VERBOSE`.
More Pattern Power

So far we've only covered a part of the features of regular expressions. In this section, we'll cover some new metacharacters, and how to use groups to retrieve portions of the text that was matched.

More Metacharacters

There are some metacharacters that we haven't covered yet. Most of them will be covered in this section.

Some of the remaining metacharacters to be discussed are zero-width assertions. They don't cause the engine to advance through the string; instead, they consume no characters at all, and simply succeed or fail. For example, \b is an assertion that the current position is located at a word boundary; the position isn't changed by the \b at all. This means that zero-width assertions should never be repeated, because if they match once at a given location, they can obviously be matched an infinite number of times.

\^ Matches at the beginning of lines. Unless the MULTILINE flag has
been set, this will only match at the beginning of the string. In **MULTILINE** mode, this also matches immediately after each newline within the string.

For example, if you wish to match the word *From* only at the beginning of a line, the RE to use is `^From`.

```python
>>> print re.search('^From', 'From Here to Eternity')
<_sre.SRE_Match object at 0x...>
```

$ Matches at the end of a line, which is defined as either the end of the string, or any location followed by a newline character.

```python
>>> print re.search('}$', '{block}')
<_sre.SRE_Match object at 0x...>
```

To match a literal `$`, use \$ or enclose it inside a character class, as in `[\$]`.

\A Matches only at the start of the string. When not in **MULTILINE** mode, \A and ^ are effectively the same. In **MULTILINE** mode, they're different: \A still matches only at the beginning of the string, but ^ may match at any location inside the string that follows a newline character.

\Z Matches only at the end of the string.

\b Word boundary. This is a zero-width assertion that matches only at the beginning or end of a word. A word is defined as a
sequence of alphanumeric characters, so the end of a word is indicated by whitespace or a non-alphanumeric character.

The following example matches class only when it's a complete word; it won't match when it's contained inside another word.

```python
>>> p = re.compile(r'\bclass\b')
>>> print p.search('no class at all')
<_sre.SRE_Match object at 0x...>
>>> print p.search('the declassified algorithm')
None
>>> print p.search('one subclass is')
None
```

There are two subtleties you should remember when using this special sequence. First, this is the worst collision between Python's string literals and regular expression sequences. In Python's string literals, \b is the backspace character, ASCII value 8. If you're not using raw strings, then Python will convert the \b to a backspace, and your RE won't match as you expect it to. The following example looks the same as our previous RE, but omits the 'r' in front of the RE string.

```python
>>> p = re.compile('\bclass\b')
>>> print p.search('no class at all')
None
>>> print p.search('\b' + 'class' + '\b')
<_sre.SRE_Match object at 0x...>
```

Second, inside a character class, where there's no use for this assertion, \b represents the backspace character, for compatibility with Python's string literals.

Another zero-width assertion, this is the opposite of \b, only matching when the current position is not at a word boundary.
Frequently you need to obtain more information than just whether the RE matched or not. Regular expressions are often used to dissect strings by writing a RE divided into several subgroups which match different components of interest. For example, an RFC-822 header line is divided into a header name and a value, separated by a ':', like this:

```
From: author@example.com
User-Agent: Thunderbird 1.5.0.9 (X11/20061227)
MIME-Version: 1.0
To: editor@example.com
```

This can be handled by writing a regular expression which matches an entire header line, and has one group which matches the header name, and another group which matches the header's value.

Groups are marked by the `(' `)` metacharacters. `(' `)` have much the same meaning as they do in mathematical expressions; they group together the expressions contained inside them, and you can repeat the contents of a group with a repeating qualifier, such as *, +, ?, or {m,n}. For example, `(ab)*` will match zero or more repetitions of `ab`.

```python
>>> p = re.compile('(ab)*')
>>> print p.match('ababababab').span()
(0, 10)
```

Groups indicated with `(' `)` also capture the starting and ending index of the text that they match; this can be retrieved by passing an argument to `group()`, `start()`, `end()`, and `span()`. Groups are numbered starting with 0. Group 0 is always present; it's the whole RE, so `match object` methods all have group 0 as their default argument. Later we'll see how to express groups that don't capture
the span of text that they match.

```python
>>> p = re.compile('(a)b')
>>> m = p.match('ab')
>>> m.group()
'ab'
>>> m.group(0)
'ab'
```

Subgroups are numbered from left to right, from 1 upward. Groups can be nested; to determine the number, just count the opening parenthesis characters, going from left to right.

```python
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

group() can be passed multiple group numbers at a time, in which case it will return a tuple containing the corresponding values for those groups.

```python
>>> m.group(2, 1, 2)
('b', 'abc', 'b')
```

The groups() method returns a tuple containing the strings for all the subgroups, from 1 up to however many there are.

```python
>>> m.groups()
('abc', 'b')
```

Backreferences in a pattern allow you to specify that the contents of an earlier capturing group must also be found at the current location in the string. For example, \1 will succeed if the exact contents of group 1 can be found at the current position, and fails otherwise.
Remember that Python's string literals also use a backslash followed by numbers to allow including arbitrary characters in a string, so be sure to use a raw string when incorporating backreferences in a RE.

For example, the following RE detects doubled words in a string.

```python
>>> p = re.compile(r'\b\w+\s+\1')
>>> p.search('Paris in the the spring').group()
'the the'
```

Backreferences like this aren't often useful for just searching through a string — there are few text formats which repeat data in this way — but you'll soon find out that they're very useful when performing string substitutions.

Non-capturing and Named Groups

Elaborate REs may use many groups, both to capture substrings of interest, and to group and structure the RE itself. In complex REs, it becomes difficult to keep track of the group numbers. There are two features which help with this problem. Both of them use a common syntax for regular expression extensions, so we'll look at that first.

Perl 5 added several additional features to standard regular expressions, and the Python `re` module supports most of them. It would have been difficult to choose new single-keystroke metacharacters or new special sequences beginning with `\` to represent the new features without making Perl's regular expressions confusingly different from standard REs. If you chose `&` as a new metacharacter, for example, old expressions would be assuming that `&` was a regular character and wouldn’t have escaped it by writing `\&` or `[^&]`.

The solution chosen by the Perl developers was to use `(？...)` as the extension syntax. `？` immediately after a parenthesis was a syntax
error because the ? would have nothing to repeat, so this didn't introduce any compatibility problems. The characters immediately after the ? indicate what extension is being used, so (?=foo) is one thing (a positive lookahead assertion) and (?:foo) is something else (a non-capturing group containing the subexpression foo).

Python adds an extension syntax to Perl's extension syntax. If the first character after the question mark is a P, you know that it's an extension that's specific to Python. Currently there are two such extensions: (?P<name>...) defines a named group, and (?P=name) is a backreference to a named group. If future versions of Perl 5 add similar features using a different syntax, the re module will be changed to support the new syntax, while preserving the Python-specific syntax for compatibility’s sake.

Now that we've looked at the general extension syntax, we can return to the features that simplify working with groups in complex REs. Since groups are numbered from left to right and a complex expression may use many groups, it can become difficult to keep track of the correct numbering. Modifying such a complex RE is annoying, too: insert a new group near the beginning and you change the numbers of everything that follows it.

Sometimes you'll want to use a group to collect a part of a regular expression, but aren't interested in retrieving the group's contents. You can make this fact explicit by using a non-capturing group: (?:. . .), where you can replace the . . . with any other regular expression.

>>> m = re.match("([abc])+", "abc")
>>> m.groups()
('c',)
>>> m = re.match("(?:[abc])+", "abc")
>>> m.groups()
()
Except for the fact that you can’t retrieve the contents of what the group matched, a non-capturing group behaves exactly the same as a capturing group; you can put anything inside it, repeat it with a repetition metacharacter such as *, and nest it within other groups (capturing or non-capturing). (?::...) is particularly useful when modifying an existing pattern, since you can add new groups without changing how all the other groups are numbered. It should be mentioned that there’s no performance difference in searching between capturing and non-capturing groups; neither form is any faster than the other.

A more significant feature is named groups: instead of referring to them by numbers, groups can be referenced by a name.

The syntax for a named group is one of the Python-specific extensions: (?P<name>...). name is, obviously, the name of the group. Named groups also behave exactly like capturing groups, and additionally associate a name with a group. The match object methods that deal with capturing groups all accept either integers that refer to the group by number or strings that contain the desired group’s name. Named groups are still given numbers, so you can retrieve information about a group in two ways:

```python
>>> p = re.compile(r'(?P<word>\b\w+\b)')
>>> m = p.search( '((( Lots of punctuation )))' )
>>> m.group('word')
'Lots'
>>> m.group(1)
'Lots'
```

Named groups are handy because they let you use easily-remembered names, instead of having to remember numbers. Here’s an example RE from the `imaplib` module:

```python
InternalDate = re.compile(r'INTERNALDATE ''
  r'(?P<day>[ 123][0-9])-(?P<mon>[A-Z][a-z][a-z])-'
  r'(?P<year>[0-9][0-9][0-9][0-9])'
)```

It's obviously much easier to retrieve \texttt{m.group('zonem')}, instead of having to remember to retrieve group 9.

The syntax for backreferences in an expression such as \texttt{(\ldots)\1} refers to the number of the group. There's naturally a variant that uses the group name instead of the number. This is another Python extension: \texttt{(?P=name)} indicates that the contents of the group called \textit{name} should again be matched at the current point. The regular expression for finding doubled words, \texttt{([^b\w]+)\s+(?P=word)} can also be written as \texttt{(?P<word>[^b\w]+)\s+(?P=word)}:

```python
>>> p = re.compile(r'(?P<word>[^b\w]+)\s+(?P=word)')
>>> p.search('Paris in the the spring').group()
'the the'
```

**Lookahead Assertions**

Another zero-width assertion is the lookahead assertion. Lookahead assertions are available in both positive and negative form, and look like this:

\texttt{(?=\ldots)}

Positive lookahead assertion. This succeeds if the contained regular expression, represented here by \texttt{\ldots}, successfully matches at the current location, and fails otherwise. But, once the contained expression has been tried, the matching engine doesn't advance at all; the rest of the pattern is tried right where the assertion started.

\texttt{(?![\ldots]}

Negative lookahead assertion. This is the opposite of the positive assertion; it succeeds if the contained expression \textit{doesn't} match
at the current position in the string.

To make this concrete, let’s look at a case where a lookahead is useful. Consider a simple pattern to match a filename and split it apart into a base name and an extension, separated by a `. `. For example, in `news.rc`, `news` is the base name, and `rc` is the filename’s extension.

The pattern to match this is quite simple:

```
.*[.].*$
```

Notice that the `.` needs to be treated specially because it’s a metacharacter; I’ve put it inside a character class. Also notice the trailing `$`; this is added to ensure that all the rest of the string must be included in the extension. This regular expression matches `foo.bar` and `autoexec.bat` and `sendmail.cf` and `printers.conf`.

Now, consider complicating the problem a bit; what if you want to match filenames where the extension is not `bat`? Some incorrect attempts:

```
.*[.][^b].*$
```

The first attempt above tries to exclude `bat` by requiring that the first character of the extension is not a `b`. This is wrong, because the pattern also doesn’t match `foo.bar`.

```
.*[.][^b][^a][^t]$```

The expression gets messier when you try to patch up the first solution by requiring one of the following cases to match: the first character of the extension isn’t `b`; the second character isn’t `a`; or the third character isn’t `t`. This accepts `foo.bar` and rejects `autoexec.bat`, but it requires a three-letter extension and won’t accept a filename with a two-letter extension such as `sendmail.cf`. 
We’ll complicate the pattern again in an effort to fix it.

\.*\.[^b]\.?.?\[^a]\.?.?..?[^t]?\)*$  

In the third attempt, the second and third letters are all made optional in order to allow matching extensions shorter than three characters, such as `sendmail.cf`.

The pattern’s getting really complicated now, which makes it hard to read and understand. Worse, if the problem changes and you want to exclude both `bat` and `exe` as extensions, the pattern would get even more complicated and confusing.

A negative lookahead cuts through all this confusion:

\.*\.[^!.bat$][^.]\)*$  
The negative lookahead means: if the expression `bat` doesn’t match at this point, try the rest of the pattern; if `bat$` does match, the whole pattern will fail. The trailing `$` is required to ensure that something like `sample.bat`, where the extension only starts with `bat`, will be allowed. The `[^.]\)*` makes sure that the pattern works when there are multiple dots in the filename.

Excluding another filename extension is now easy; simply add it as an alternative inside the assertion. The following pattern excludes filenames that end in either `bat` or `exe`:

\.*\.[^!.bat$|exe$][^.]\)*$
Modifying Strings

Up to this point, we've simply performed searches against a static string. Regular expressions are also commonly used to modify strings in various ways, using the following pattern methods:

<table>
<thead>
<tr>
<th>Method/Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>split()</td>
<td>Split the string into a list, splitting it wherever the RE matches</td>
</tr>
<tr>
<td>sub()</td>
<td>Find all substrings where the RE matches, and replace them with a different string</td>
</tr>
<tr>
<td>subn()</td>
<td>Does the same thing as sub(), but returns the new string and the number of replacements</td>
</tr>
</tbody>
</table>

Splitting Strings

The split() method of a pattern splits a string apart wherever the RE matches, returning a list of the pieces. It's similar to the split() method of strings but provides much more generality in the delimiters that you can split by; split() only supports splitting by whitespace or by a fixed string. As you'd expect, there's a module-level re.split() function, too.

```
.split(string[, maxsplit=0])
```

Split string by the matches of the regular expression. If capturing parentheses are used in the RE, then their contents will also be returned as part of the resulting list. If maxsplit is nonzero, at most maxsplit splits are performed.

You can limit the number of splits made, by passing a value for maxsplit. When maxsplit is nonzero, at most maxsplit splits will be made, and the remainder of the string is returned as the final element of the list. In the following example, the delimiter is any...
sequence of non-alphanumeric characters.

```python
>>> p = re.compile(r'\W+')
>>> p.split('This is a test, short and sweet, of split().')
['This', 'is', 'a', 'test', 'short', 'and', 'sweet', 'of', 'split()']
>>> p.split('This is a test, short and sweet, of split().', 3)
['This', 'is', 'a', 'test, short and sweet, of split().']
```

Sometimes you’re not only interested in what the text between delimiters is, but also need to know what the delimiter was. If capturing parentheses are used in the RE, then their values are also returned as part of the list. Compare the following calls:

```python
>>> p = re.compile(r'\W+')
>>> p2 = re.compile(r'([^\W]+)')
>>> p.split('This... is a test.')
['This', 'is', 'a', 'test', '']
>>> p2.split('This... is a test.')
['This', '...', 'is', '', 'a', '', 'test', '.', '']
```

The module-level function `re.split()` adds the RE to be used as the first argument, but is otherwise the same.

```python
>>> re.split('([^\W]+)', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split('([^\W]+)', 'Words, words, words.')
['Words', '', '', 'words', '', 'words', '.', '', '']
>>> re.split('([^\W]+)', 'Words, words, words.', 1)
['Words', 'words, words. ']
```

**Search and Replace**

Another common task is to find all the matches for a pattern, and replace them with a different string. The `sub()` method takes a replacement value, which can be either a string or a function, and the string to be processed.

```python
sub(replacement, string[, count=0])
```
Returns the string obtained by replacing the leftmost non-overlapping occurrences of the RE in string by the replacement replacement. If the pattern isn’t found, string is returned unchanged.

The optional argument count is the maximum number of pattern occurrences to be replaced; count must be a non-negative integer. The default value of 0 means to replace all occurrences.

Here’s a simple example of using the sub() method. It replaces colour names with the word colour:

```python
>>> p = re.compile('(blue|white|red)')
>>> p.sub('colour', 'blue socks and red shoes')
'colour socks and colour shoes'
>>> p.sub('colour', 'blue socks and red shoes', count=1)
'colour socks and red shoes'
```

The subn() method does the same work, but returns a 2-tuple containing the new string value and the number of replacements that were performed:

```python
>>> p = re.compile('(blue|white|red)')
>>> p.subn('colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
>>> p.subn('colour', 'no colours at all')
('no colours at all', 0)
```

Empty matches are replaced only when they’re not adjacent to a previous match.

```python
>>> p = re.compile('x*')
>>> p.sub('-', 'abxd')
'-a-b-d-'
```

If replacement is a string, any backslash escapes in it are processed. That is, `\n` is converted to a single newline character, `\r` is converted
to a carriage return, and so forth. Unknown escapes such as \j are left alone. Backreferences, such as \6, are replaced with the substring matched by the corresponding group in the RE. This lets you incorporate portions of the original text in the resulting replacement string.

This example matches the word `section` followed by a string enclosed in `{, }`, and changes `section` to `subsection`:

```python
>>> p = re.compile('section{ ( [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}', 'section{First} section{Second}')
'subsection{First} subsection{Second}'
```

There's also a syntax for referring to named groups as defined by the `(\?P<name>...)` syntax. `\g<name>` will use the substring matched by the group named `name`, and `\g<number>` uses the corresponding group number. `\g<2>` is therefore equivalent to `\2`, but isn't ambiguous in a replacement string such as `\g<2>0`. (`\20` would be interpreted as a reference to group 20, not a reference to group 2 followed by the literal character '0'.) The following substitutions are all equivalent, but use all three variations of the replacement string.

```python
>>> p = re.compile('section{ (?P<name> [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}','section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<1>}','section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<name>}','section{First}')
'subsection{First}'
```

*replacement* can also be a function, which gives you even more control. If *replacement* is a function, the function is called for every non-overlapping occurrence of *pattern*. On each call, the function is passed a *match object* argument for the match and can use this information to compute the desired replacement string and return it.
In the following example, the replacement function translates decimals into hexadecimal:

```python
>>> def hexrepl(match):
...    "Return the hex string for a decimal number"
...    value = int(match.group())
...    return hex(value)
...
>>> p = re.compile(r'\d+')
>>> p.sub(hexrepl, 'Call 65490 for printing, 49152 for user code.
Call 0xfffd2 for printing, 0xc000 for user code.')
```

When using the module-level `re.sub()` function, the pattern is passed as the first argument. The pattern may be provided as an object or as a string; if you need to specify regular expression flags, you must either use a pattern object as the first parameter, or use embedded modifiers in the pattern string, e.g. `sub("(?i)b+", "x", "bbbb BBBB")` returns `x x`.
Common Problems

Regular expressions are a powerful tool for some applications, but in some ways their behaviour isn’t intuitive and at times they don’t behave the way you may expect them to. This section will point out some of the most common pitfalls.

Use String Methods

Sometimes using the `re` module is a mistake. If you’re matching a fixed string, or a single character class, and you’re not using any `re` features such as the `IGNORECASE` flag, then the full power of regular expressions may not be required. Strings have several methods for performing operations with fixed strings and they’re usually much faster, because the implementation is a single small C loop that’s been optimized for the purpose, instead of the large, more generalized regular expression engine.

One example might be replacing a single fixed string with another one; for example, you might replace `word` with `deed`. `re.sub()` seems like the function to use for this, but consider the `replace()` method. Note that `replace()` will also replace `word` inside words, turning `swordfish` into `sdeedfish`, but the naive RE `word` would have done that, too. (To avoid performing the substitution on parts of words, the pattern would have to be `\bword\b`, in order to require that `word` have a word boundary on either side. This takes the job beyond `replace()`'s abilities.)

Another common task is deleting every occurrence of a single character from a string or replacing it with another single character. You might do this with something like `re.sub('\n', '', S)`, but `translate()` is capable of doing both tasks and will be faster than
any regular expression operation can be.

In short, before turning to the `re` module, consider whether your problem can be solved with a faster and simpler string method.

## match() versus search()

The `match()` function only checks if the RE matches at the beginning of the string while `search()` will scan forward through the string for a match. It’s important to keep this distinction in mind. Remember, `match()` will only report a successful match which will start at 0; if the match wouldn’t start at zero, `match()` will *not* report it.

```python
>>> print re.match('super', 'superstition').span()
(0, 5)
>>> print re.match('super', 'insuperable')
None
```

On the other hand, `search()` will scan forward through the string, reporting the first match it finds.

```python
>>> print re.search('super', 'superstition').span()
(0, 5)
>>> print re.search('super', 'insuperable').span()
(2, 7)
```

Sometimes you’ll be tempted to keep using `re.match()`, and just add `.*` to the front of your RE. Resist this temptation and use `re.search()` instead. The regular expression compiler does some analysis of REs in order to speed up the process of looking for a match. One such analysis figures out what the first character of a match must be; for example, a pattern starting with `Crow` must match starting with a `'c'`. The analysis lets the engine quickly scan through the string looking for the starting character, only trying the full match if a `'c'` is found.
Adding .* defeats this optimization, requiring scanning to the end of the string and then backtracking to find a match for the rest of the RE. Use re.search() instead.

## Greedy versus Non-Greedy

When repeating a regular expression, as in a*, the resulting action is to consume as much of the pattern as possible. This fact often bites you when you’re trying to match a pair of balanced delimiters, such as the angle brackets surrounding an HTML tag. The naive pattern for matching a single HTML tag doesn’t work because of the greedy nature of .*.

```python
>>> s = '<html><head><title>Title</title>'
>>> len(s)
32
>>> print re.match('.*', s).span()
(0, 32)
>>> print re.match('.*', s).group()
<html><head><title>Title</title>
```

The RE matches the < in <html>, and the .* consumes the rest of the string. There’s still more left in the RE, though, and the > can’t match at the end of the string, so the regular expression engine has to backtrack character by character until it finds a match for the >. The final match extends from the < in <html> to the > in </title>, which isn’t what you want.

In this case, the solution is to use the non-greedy qualifiers *?, +?, ??, or {m,n}?, which match as little text as possible. In the above example, the > is tried immediately after the first < matches, and when it fails, the engine advances a character at a time, retrying the > at every step. This produces just the right result:

```python
>>> print re.match('.*?', s).group()
<html><head><title>Title</title>
```
(Note that parsing HTML or XML with regular expressions is painful. Quick-and-dirty patterns will handle common cases, but HTML and XML have special cases that will break the obvious regular expression; by the time you've written a regular expression that handles all of the possible cases, the patterns will be very complicated. Use an HTML or XML parser module for such tasks.)

Using `re.VERBOSE`

By now you've probably noticed that regular expressions are a very compact notation, but they're not terribly readable. REs of moderate complexity can become lengthy collections of backslashes, parentheses, and metacharacters, making them difficult to read and understand.

For such REs, specifying the `re.VERBOSE` flag when compiling the regular expression can be helpful, because it allows you to format the regular expression more clearly.

The `re.VERBOSE` flag has several effects. Whitespace in the regular expression that isn't inside a character class is ignored. This means that an expression such as `dog | cat` is equivalent to the less readable `dog|cat`, but `[a b]` will still match the characters 'a', 'b', or a space. In addition, you can also put comments inside a RE; comments extend from a '#' character to the next newline. When used with triple-quoted strings, this enables REs to be formatted more neatly:

```python
pat = re.compile(r'''\s* (?P<header>[^:]+) \s* : (?P<value>.*?) # Skip leading whitespace # Header name # Whitespace, and a colon # The header's value -- *? used to # lose the following trailing whitespace'''
``
This is far more readable than:

```python
pat = re.compile(r'^\s*(?P<header>[^:]+)\s*: (?P<value>.*?)\s*$')
```
Regular expressions are a complicated topic. Did this document help you understand them? Were there parts that were unclear, or Problems you encountered that weren’t covered here? If so, please send suggestions for improvements to the author.

The most complete book on regular expressions is almost certainly Jeffrey Friedl's Mastering Regular Expressions, published by O'Reilly. Unfortunately, it exclusively concentrates on Perl and Java’s flavours of regular expressions, and doesn’t contain any Python material at all, so it won’t be useful as a reference for programming in Python. (The first edition covered Python's now-removed `regex` module, which won't help you much.) Consider checking it out from your library.

Footnotes

[1] Introduced in Python 2.2.2.
Abstract

Sockets are used nearly everywhere, but are one of the most severely misunderstood technologies around. This is a 10,000 foot overview of sockets. It's not really a tutorial - you'll still have work to do in getting things operational. It doesn't cover the fine points (and there are a lot of them), but I hope it will give you enough background to begin using them decently.
Sockets

I'm only going to talk about INET sockets, but they account for at least 99% of the sockets in use. And I'll only talk about STREAM sockets - unless you really know what you're doing (in which case this HOWTO isn't for you!), you'll get better behavior and performance from a STREAM socket than anything else. I will try to clear up the mystery of what a socket is, as well as some hints on how to work with blocking and non-blocking sockets. But I'll start by talking about blocking sockets. You'll need to know how they work before dealing with non-blocking sockets.

Part of the trouble with understanding these things is that “socket” can mean a number of subtly different things, depending on context. So first, let’s make a distinction between a “client” socket - an endpoint of a conversation, and a “server” socket, which is more like a switchboard operator. The client application (your browser, for example) uses “client” sockets exclusively; the web server it’s talking to uses both “server” sockets and “client” sockets.

History

Of the various forms of IPC, sockets are by far the most popular. On any given platform, there are likely to be other forms of IPC that are faster, but for cross-platform communication, sockets are about the only game in town.

They were invented in Berkeley as part of the BSD flavor of Unix. They spread like wildfire with the Internet. With good reason — the combination of sockets with INET makes talking to arbitrary machines around the world unbelievably easy (at least compared to other schemes).
Creating a Socket

Roughly speaking, when you clicked on the link that brought you to this page, your browser did something like the following:

```python
# create an INET, STREAMing socket
s = socket.socket(
    socket.AF_INET, socket.SOCK_STREAM)
# now connect to the web server on port 80
# - the normal http port
s.connect(("www.mcmillan-inc.com", 80))
```

When the `connect` completes, the socket `s` can be used to send in a request for the text of the page. The same socket will read the reply, and then be destroyed. That’s right, destroyed. Client sockets are normally only used for one exchange (or a small set of sequential exchanges).

What happens in the web server is a bit more complex. First, the web server creates a “server socket”:

```python
# create an INET, STREAMing socket
serversocket = socket.socket(
    socket.AF_INET, socket.SOCK_STREAM)
# bind the socket to a public host,
# and a well-known port
serversocket.bind((socket.gethostname(), 80))
# become a server socket
serversocket.listen(5)
```

A couple things to notice: we used `socket.gethostname()` so that the socket would be visible to the outside world. If we had used `s.bind(('localhost', 80))` or `s.bind(('127.0.0.1', 80))` we would still have a “server” socket, but one that was only visible within the same machine. `s.bind(('', 80))` specifies that the socket is reachable by any address the machine happens to have.
A second thing to note: low number ports are usually reserved for “well known” services (HTTP, SNMP etc). If you're playing around, use a nice high number (4 digits).

Finally, the argument to `listen` tells the socket library that we want it to queue up as many as 5 connect requests (the normal max) before refusing outside connections. If the rest of the code is written properly, that should be plenty.

Now that we have a “server” socket, listening on port 80, we can enter the mainloop of the web server:

```python
while 1:
    # accept connections from outside
    (clientsocket, address) = serversocket.accept()
    # now do something with the clientsocket
    # in this case, we'll pretend this is a threaded server
    ct = client_thread(clientsocket)
    ct.run()
```

There's actually 3 general ways in which this loop could work - dispatching a thread to handle `clientsocket`, create a new process to handle `clientsocket`, or restructure this app to use non-blocking sockets, and multiplex between our “server” socket and any active `clientsocket`s using `select`. More about that later. The important thing to understand now is this: this is all a “server” socket does. It doesn’t send any data. It doesn’t receive any data. It just produces “client” sockets. Each `clientsocket` is created in response to some other “client” socket doing a `connect()` to the host and port we’re bound to. As soon as we’ve created that `clientsocket`, we go back to listening for more connections. The two “clients” are free to chat it up - they are using some dynamically allocated port which will be recycled when the conversation ends.
If you need fast IPC between two processes on one machine, you should look into whatever form of shared memory the platform offers. A simple protocol based around shared memory and locks or semaphores is by far the fastest technique.

If you do decide to use sockets, bind the “server” socket to 'localhost'. On most platforms, this will take a shortcut around a couple of layers of network code and be quite a bit faster.
Using a Socket

The first thing to note, is that the web browser’s “client” socket and the web server’s “client” socket are identical beasts. That is, this is a “peer to peer” conversation. Or to put it another way, as the designer, you will have to decide what the rules of etiquette are for a conversation. Normally, the connecting socket starts the conversation, by sending in a request, or perhaps a signon. But that’s a design decision - it’s not a rule of sockets.

Now there are two sets of verbs to use for communication. You can use send and recv, or you can transform your client socket into a file-like beast and use read and write. The latter is the way Java presents its sockets. I’m not going to talk about it here, except to warn you that you need to use flush on sockets. These are buffered “files”, and a common mistake is to write something, and then read for a reply. Without a flush in there, you may wait forever for the reply, because the request may still be in your output buffer.

Now we come to the major stumbling block of sockets - send and recv operate on the network buffers. They do not necessarily handle all the bytes you hand them (or expect from them), because their major focus is handling the network buffers. In general, they return when the associated network buffers have been filled (send) or emptied (recv). They then tell you how many bytes they handled. It is your responsibility to call them again until your message has been completely dealt with.

When a recv returns 0 bytes, it means the other side has closed (or is in the process of closing) the connection. You will not receive any more data on this connection. Ever. You may be able to send data successfully; I’ll talk more about this later.
A protocol like HTTP uses a socket for only one transfer. The client sends a request, then reads a reply. That's it. The socket is discarded. This means that a client can detect the end of the reply by receiving 0 bytes.

But if you plan to reuse your socket for further transfers, you need to realize that there is no EOT on a socket. I repeat: if a socket send or recv returns after handling 0 bytes, the connection has been broken. If the connection has not been broken, you may wait on a recv forever, because the socket will not tell you that there's nothing more to read (for now). Now if you think about that a bit, you'll come to realize a fundamental truth of sockets: messages must either be fixed length (yuck), or be delimited (shrug), or indicate how long they are (much better), or end by shutting down the connection. The choice is entirely yours, (but some ways are righter than others).

Assuming you don't want to end the connection, the simplest solution is a fixed length message:

```python
class mysocket:
    '''demonstration class only
    - coded for clarity, not efficiency
    '''

    def __init__(self, sock=None):
        if sock is None:
            self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        else:
            self.sock = sock

    def connect(self, host, port):
        self.sock.connect((host, port))

    def mysend(self, msg):
        totalsent = 0
        while totalsent < MSGLEN:
            sent = self.sock.send(msg[totalsent:]
            if sent == 0:
                raise RuntimeError("socket connection broken")
```
```python
totalsent = totalsent + sent

def myreceive(self):
    chunks = []
    bytes_recd = 0
    while bytes_recd < MSGLEN:
        chunk = self.sock.recv(min(MSGLEN - bytes_recd, 204)
        if chunk == '':
            raise RuntimeError("socket connection broken")
        chunks.append(chunk)
        bytes_recd = bytes_recd + len(chunk)
    return ''.join(chunks)
```

The sending code here is usable for almost any messaging scheme - in Python you send strings, and you can use `len()` to determine its length (even if it has embedded `\0` characters). It's mostly the receiving code that gets more complex. (And in C, it's not much worse, except you can't use `strlen` if the message has embedded `\0`s.)

The easiest enhancement is to make the first character of the message an indicator of message type, and have the type determine the length. Now you have two `recv`s - the first to get (at least) that first character so you can look up the length, and the second in a loop to get the rest. If you decide to go the delimited route, you'll be receiving in some arbitrary chunk size, (4096 or 8192 is frequently a good match for network buffer sizes), and scanning what you've received for a delimiter.

One complication to be aware of: if your conversational protocol allows multiple messages to be sent back to back (without some kind of reply), and you pass `recv` an arbitrary chunk size, you may end up reading the start of a following message. You'll need to put that aside and hold onto it, until it's needed.

Prefixing the message with its length (say, as 5 numeric characters) gets more complex, because (believe it or not), you may not get all 5
characters in one `recv`. In playing around, you'll get away with it; but in high network loads, your code will very quickly break unless you use two `recv` loops - the first to determine the length, the second to get the data part of the message. Nasty. This is also when you'll discover that `send` does not always manage to get rid of everything in one pass. And despite having read this, you will eventually get bit by it!

In the interests of space, building your character, (and preserving my competitive position), these enhancements are left as an exercise for the reader. Lets move on to cleaning up.

**Binary Data**

It is perfectly possible to send binary data over a socket. The major problem is that not all machines use the same formats for binary data. For example, a Motorola chip will represent a 16 bit integer with the value 1 as the two hex bytes 00 01. Intel and DEC, however, are byte-reversed - that same 1 is 01 00. Socket libraries have calls for converting 16 and 32 bit integers - `ntohl`, `htonl`, `ntohs`, `htons` where “n” means `network` and “h” means `host`, “s” means `short` and “l” means `long`. Where network order is host order, these do nothing, but where the machine is byte-reversed, these swap the bytes around appropriately.

In these days of 32 bit machines, the ascii representation of binary data is frequently smaller than the binary representation. That's because a surprising amount of the time, all those longs have the value 0, or maybe 1. The string “0” would be two bytes, while binary is four. Of course, this doesn't fit well with fixed-length messages. Decisions, decisions.
Disconnected

Strictly speaking, you're supposed to use `shutdown` on a socket before you `close` it. The `shutdown` is an advisory to the socket at the other end. Depending on the argument you pass it, it can mean “I'm not going to send anymore, but I'll still listen”, or “I'm not listening, good riddance!”. Most socket libraries, however, are so used to programmers neglecting to use this piece of etiquette that normally a `close` is the same as `shutdown(); close()`. So in most situations, an explicit `shutdown` is not needed.

One way to use `shutdown` effectively is in an HTTP-like exchange. The client sends a request and then does a `shutdown(1)`. This tells the server “This client is done sending, but can still receive.” The server can detect “EOF” by a receive of 0 bytes. It can assume it has the complete request. The server sends a reply. If the `send` completes successfully then, indeed, the client was still receiving.

Python takes the automatic shutdown a step further, and says that when a socket is garbage collected, it will automatically do a `close` if it's needed. But relying on this is a very bad habit. If your socket just disappears without doing a `close`, the socket at the other end may hang indefinitely, thinking you're just being slow. *Please close* your sockets when you're done.

**When Sockets Die**

Probably the worst thing about using blocking sockets is what happens when the other side comes down hard (without doing a `close`). Your socket is likely to hang. SOCKSTREAM is a reliable protocol, and it will wait a long, long time before giving up on a connection. If you're using threads, the entire thread is essentially
dead. There’s not much you can do about it. As long as you aren’t doing something dumb, like holding a lock while doing a blocking read, the thread isn’t really consuming much in the way of resources. Do not try to kill the thread - part of the reason that threads are more efficient than processes is that they avoid the overhead associated with the automatic recycling of resources. In other words, if you do manage to kill the thread, your whole process is likely to be screwed up.
Non-blocking Sockets

If you've understood the preceding, you already know most of what you need to know about the mechanics of using sockets. You'll still use the same calls, in much the same ways. It's just that, if you do it right, your app will be almost inside-out.

In Python, you use `socket.setblocking(0)` to make it non-blocking. In C, it's more complex, (for one thing, you'll need to choose between the BSD flavor `O_NONBLOCK` and the almost indistinguishable Posix flavor `O_NDELAY`, which is completely different from `TCP_NODELAY`), but it's the exact same idea. You do this after creating the socket, but before using it. (Actually, if you're nuts, you can switch back and forth.)

The major mechanical difference is that `send`, `recv`, `connect` and `accept` can return without having done anything. You have (of course) a number of choices. You can check return code and error codes and generally drive yourself crazy. If you don't believe me, try it sometime. Your app will grow large, buggy and suck CPU. So let's skip the brain-dead solutions and do it right.

Use `select`.

In C, coding `select` is fairly complex. In Python, it's a piece of cake, but it's close enough to the C version that if you understand `select` in Python, you'll have little trouble with it in C:

```python
ready_to_read, ready_to_write, in_error = 
    select.select( 
        potential_readers, 
        potential_writers, 
        potential_errs, 
        timeout)
```
You pass `select` three lists: the first contains all sockets that you might want to try reading; the second all the sockets you might want to try writing to, and the last (normally left empty) those that you want to check for errors. You should note that a socket can go into more than one list. The `select` call is blocking, but you can give it a timeout. This is generally a sensible thing to do - give it a nice long timeout (say a minute) unless you have good reason to do otherwise.

In return, you will get three lists. They contain the sockets that are actually readable, writable and in error. Each of these lists is a subset (possibly empty) of the corresponding list you passed in.

If a socket is in the output readable list, you can be as-close-to-certain-as-we-ever-get-in-this-business that a `recv` on that socket will return something. Same idea for the writable list. You'll be able to send something. Maybe not all you want to, but something is better than nothing. (Actually, any reasonably healthy socket will return as writable - it just means outbound network buffer space is available.)

If you have a “server” socket, put it in the potential_readers list. If it comes out in the readable list, your `accept` will (almost certainly) work. If you have created a new socket to connect to someone else, put it in the potential_writers list. If it shows up in the writable list, you have a decent chance that it has connected.

One very nasty problem with `select`: if somewhere in those input lists of sockets is one which has died a nasty death, the `select` will fail. You then need to loop through every single damn socket in all those lists and do a `select([sock],[[],[],0])` until you find the bad one. That timeout of 0 means it won’t take long, but it’s ugly.

Actually, `select` can be handy even with blocking sockets. It’s one way of determining whether you will block - the socket returns as
readable when there's something in the buffers. However, this still doesn't help with the problem of determining whether the other end is done, or just busy with something else.

**Portability alert**: On Unix, `select` works both with the sockets and files. Don't try this on Windows. On Windows, `select` works with sockets only. Also note that in C, many of the more advanced socket options are done differently on Windows. In fact, on Windows I usually use threads (which work very, very well) with my sockets. Face it, if you want any kind of performance, your code will look very different on Windows than on Unix.

**Performance**

There's no question that the fastest sockets code uses non-blocking sockets and `select` to multiplex them. You can put together something that will saturate a LAN connection without putting any strain on the CPU. The trouble is that an app written this way can't do much of anything else - it needs to be ready to shuffle bytes around at all times.

Assuming that your app is actually supposed to do something more than that, threading is the optimal solution, (and using non-blocking sockets will be faster than using blocking sockets). Unfortunately, threading support in Unixes varies both in API and quality. So the normal Unix solution is to fork a subprocess to deal with each connection. The overhead for this is significant (and don't do this on Windows - the overhead of process creation is enormous there). It also means that unless each subprocess is completely independent, you'll need to use another form of IPC, say a pipe, or shared memory and semaphores, to communicate between the parent and child processes.

Finally, remember that even though blocking sockets are somewhat slower than non-blocking, in many cases they are the “right” solution.
After all, if your app is driven by the data it receives over a socket, there’s not much sense in complicating the logic just so your app can wait on `select` instead of `recv`.
Python lists have a built-in `list.sort()` method that modifies the list in-place. There is also a `sorted()` built-in function that builds a new sorted list from an iterable.

In this document, we explore the various techniques for sorting data using Python.
A simple ascending sort is very easy: just call the `sorted()` function. It returns a new sorted list:

```python
>>> sorted([5, 2, 3, 1, 4])
[1, 2, 3, 4, 5]
```

You can also use the `list.sort()` method of a list. It modifies the list in-place (and returns `None` to avoid confusion). Usually it’s less convenient than `sorted()` - but if you don’t need the original list, it’s slightly more efficient.

```python
>>> a = [5, 2, 3, 1, 4]
>>> a.sort()
>>> a
[1, 2, 3, 4, 5]
```

Another difference is that the `list.sort()` method is only defined for lists. In contrast, the `sorted()` function accepts any iterable.

```python
[1, 2, 3, 4, 5]
```
Key Functions

Starting with Python 2.4, both `list.sort()` and `sorted()` added a `key` parameter to specify a function to be called on each list element prior to making comparisons.

For example, here's a case-insensitive string comparison:

```python
>>> sorted("This is a test string from Andrew".split(), key=str.lower)
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```

The value of the `key` parameter should be a function that takes a single argument and returns a key to use for sorting purposes. This technique is fast because the key function is called exactly once for each input record.

A common pattern is to sort complex objects using some of the object's indices as keys. For example:

```python
>>> student_tuples = [
...   ('john', 'A', 15),
...   ('jane', 'B', 12),
...   ('dave', 'B', 10),
... ]
>>> sorted(student_tuples, key=lambda student: student[2])  # sort by age
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

The same technique works for objects with named attributes. For example:

```python
>>> class Student:
...   def __init__(self, name, grade, age):
...     self.name = name
...     self.grade = grade
...     self.age = age
...   def __repr__(self):
```
... return repr((self.name, self.grade, self.age))

>>> student_objects = [
...     Student('john', 'A', 15),
...     Student('jane', 'B', 12),
...     Student('dave', 'B', 10),
... ]

>>> sorted(student_objects, key=lambda student: student.age)
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
Operator Module Functions

The key-function patterns shown above are very common, so Python provides convenience functions to make accessor functions easier and faster. The operator module has `operator.itemgetter()`, `operator.attrgetter()`, and starting in Python 2.5 an `operator.methodcaller()` function.

Using those functions, the above examples become simpler and faster:

```python
>>> from operator import itemgetter, attrgetter

>>> sorted(student_tuples, key=itemgetter(2))
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

>>> sorted(student_objects, key=attrgetter('age'))
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

The operator module functions allow multiple levels of sorting. For example, to sort by `grade` then by `age`:

```python
>>> sorted(student_tuples, key=itemgetter(1,2))
[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]

>>> sorted(student_objects, key=attrgetter('grade', 'age'))
[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]
```

The `operator.methodcaller()` function makes method calls with fixed parameters for each object being sorted. For example, the `str.count()` method could be used to compute message priority by counting the number of exclamation marks in a message:

```python
>>> from operator import methodcaller

>>> messages = ['critical!!!', 'hurry!', 'standby', 'immediate!']
>>> messages.sort(key=methodcaller('count', '!'))
```

```python
[('immediate!', 1), ('hurry!', 1), ('critical!!!', 3), ('standby', 0)]
```
>>> sorted(messages, key=methodcaller('count', '!''))
['standby', 'hurry!', 'immediate!!', 'critical!!']
Ascending and Descending

Both `list.sort()` and `sorted()` accept a `reverse` parameter with a boolean value. This is used to flag descending sorts. For example, to get the student data in reverse age order:

```python
>>> sorted(student_tuples, key=itemgetter(2), reverse=True)
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```

```python
>>> sorted(student_objects, key=attrgetter('age'), reverse=True)
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```
Sort Stability and Complex Sorts

Starting with Python 2.2, sorts are guaranteed to be stable. That means that when multiple records have the same key, their original order is preserved.

```python
>>> data = [('red', 1), ('blue', 1), ('red', 2), ('blue', 2)]
>>> sorted(data, key=itemgetter(0))
[('blue', 1), ('blue', 2), ('red', 1), ('red', 2)]
```

Notice how the two records for blue retain their original order so that ('blue', 1) is guaranteed to precede ('blue', 2).

This wonderful property lets you build complex sorts in a series of sorting steps. For example, to sort the student data by descending grade and then ascending age, do the age sort first and then sort again using grade:

```python
>>> s = sorted(student_objects, key=attrgetter('age'))  # so
>>> sorted(s, key=attrgetter('grade'), reverse=True)  # no
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

The Timsort algorithm used in Python does multiple sorts efficiently because it can take advantage of any ordering already present in a dataset.
The Old Way Using Decorate-Sort-Undecorate

This idiom is called Decorate-Sort-Undecorate after its three steps:

- First, the initial list is decorated with new values that control the sort order.
- Second, the decorated list is sorted.
- Finally, the decorations are removed, creating a list that contains only the initial values in the new order.

For example, to sort the student data by grade using the DSU approach:

```python
>>> decorated = [(student.grade, i, student) for i, student in enumerate(students)]
>>> decorated.sort()
>>> [student for grade, i, student in decorated]
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```

This idiom works because tuples are compared lexicographically; the first items are compared; if they are the same then the second items are compared, and so on.

It is not strictly necessary in all cases to include the index \( i \) in the decorated list, but including it gives two benefits:

- The sort is stable – if two items have the same key, their order will be preserved in the sorted list.
- The original items do not have to be comparable because the ordering of the decorated tuples will be determined by at most the first two items. So for example the original list could contain complex numbers which cannot be sorted directly.

Another name for this idiom is Schwartzian transform, after Randal
L. Schwartz, who popularized it among Perl programmers.

For large lists and lists where the comparison information is expensive to calculate, and Python versions before 2.4, DSU is likely to be the fastest way to sort the list. For 2.4 and later, key functions provide the same functionality.
The Old Way Using the *cmp* Parameter

Many constructs given in this HOWTO assume Python 2.4 or later. Before that, there was no `sorted()` builtin and `list.sort()` took no keyword arguments. Instead, all of the Py2.x versions supported a *cmp* parameter to handle user specified comparison functions.

In Python 3, the *cmp* parameter was removed entirely (as part of a larger effort to simplify and unify the language, eliminating the conflict between rich comparisons and the `__cmp__()` magic method).

In Python 2, `sort()` allowed an optional function which can be called for doing the comparisons. That function should take two arguments to be compared and then return a negative value for less-than, return zero if they are equal, or return a positive value for greater-than. For example, we can do:

```python
>>> def numeric_compare(x, y):
...     return x - y
>>> sorted([5, 2, 4, 1, 3], cmp=numeric_compare)
[1, 2, 3, 4, 5]
```

Or you can reverse the order of comparison with:

```python
>>> def reverse_numeric(x, y):
...     return x - y
>>> sorted([5, 2, 4, 1, 3], cmp=reverse_numeric)
[5, 4, 3, 2, 1]
```

When porting code from Python 2.x to 3.x, the situation can arise when you have the user supplying a comparison function and you need to convert that to a key function. The following wrapper makes that easy to do:

```python
def cmp_to_key(mycmp):
    'Convert a cmp= function into a key= function'
```
class K(object):
    def __init__(self, obj, *args):
        self.obj = obj
    def __lt__(self, other):
        return mycmp(self.obj, other.obj) < 0
    def __gt__(self, other):
        return mycmp(self.obj, other.obj) > 0
    def __eq__(self, other):
        return mycmp(self.obj, other.obj) == 0
    def __le__(self, other):
        return mycmp(self.obj, other.obj) <= 0
    def __ge__(self, other):
        return mycmp(self.obj, other.obj) >= 0
    def __ne__(self, other):
        return mycmp(self.obj, other.obj) != 0
    return K

To convert to a key function, just wrap the old comparison function:

>>> sorted([5, 2, 4, 1, 3], key=cmp_to_key(reverse_numeric))
[5, 4, 3, 2, 1]

In Python 2.7, the functools.cmp_to_key() function was added to the functools module.
Odd and Ends

- For locale aware sorting, use `locale.strxfrm()` for a key function or `locale.strcoll()` for a comparison function.

- The `reverse` parameter still maintains sort stability (so that records with equal keys retain their original order). Interestingly, that effect can be simulated without the parameter by using the builtin `reversed()` function twice:

```python
>>> data = [('red', 1), ('blue', 1), ('red', 2), ('blue', 2)
>>> standard_way = sorted(data, key=itemgetter(0), reverse=True)
>>> double_reversed = list(reversed(sorted(reversed(data), key=itemgetter(0))))
>>> assert standard_way == double_reversed
>>> standard_way
[('red', 1), ('red', 2), ('blue', 1), ('blue', 2)]
```

- To create a standard sort order for a class, just add the appropriate rich comparison methods:

```python
>>> Student.__eq__ = lambda self, other: self.age == other.age
>>> Student.__ne__ = lambda self, other: self.age != other.age
>>> Student.__lt__ = lambda self, other: self.age < other.age
>>> Student.__le__ = lambda self, other: self.age <= other.age
>>> Student.__gt__ = lambda self, other: self.age > other.age
>>> Student.__ge__ = lambda self, other: self.age >= other.age
>>> sorted(student_objects)
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

For general purpose comparisons, the recommended approach is to define all six rich comparison operators. The `functools.total_ordering()` class decorator makes this easy to implement.

- Key functions need not depend directly on the objects being sorted. A key function can also access external resources. For
instance, if the student grades are stored in a dictionary, they can be used to sort a separate list of student names:

```python
>>> students = ['dave', 'john', 'jane']
>>> grades = {'john': 'F', 'jane': 'A', 'dave': 'C'}
>>> sorted(students, key=grades.__getitem__)
['jane', 'dave', 'john']
```
This HOWTO discusses Python 2.x's support for Unicode, and explains various problems that people commonly encounter when trying to work with Unicode. For the Python 3 version, see <https://docs.python.org/3/howto/unicode.html>. 
Introduction to Unicode

History of Character Codes

In 1968, the American Standard Code for Information Interchange, better known by its acronym ASCII, was standardized. ASCII defined numeric codes for various characters, with the numeric values running from 0 to 127. For example, the lowercase letter ‘a’ is assigned 97 as its code value.

ASCII was an American-developed standard, so it only defined unaccented characters. There was an ‘e’, but no ‘é’ or ‘ï’. This meant that languages which required accented characters couldn’t be faithfully represented in ASCII. (Actually the missing accents matter for English, too, which contains words such as ‘naïve’ and ‘café’, and some publications have house styles which require spellings such as ‘coöperate’.)

For a while people just wrote programs that didn’t display accents. I remember looking at Apple ][ BASIC programs, published in French-language publications in the mid-1980s, that had lines like these:

```
PRINT "MISE A JOUR TERMINEE"
PRINT "PARAMETRES ENREGISTRES"
```

Those messages should contain accents, and they just look wrong to someone who can read French.

In the 1980s, almost all personal computers were 8-bit, meaning that bytes could hold values ranging from 0 to 255. ASCII codes only went up to 127, so some machines assigned values between 128 and 255 to accented characters. Different machines had different codes, however, which led to problems exchanging files. Eventually various commonly used sets of values for the 128–255 range
emerged. Some were true standards, defined by the International Standards Organization, and some were de facto conventions that were invented by one company or another and managed to catch on.

255 characters aren’t very many. For example, you can’t fit both the accented characters used in Western Europe and the Cyrillic alphabet used for Russian into the 128–255 range because there are more than 128 such characters.

You could write files using different codes (all your Russian files in a coding system called KOI8, all your French files in a different coding system called Latin1), but what if you wanted to write a French document that quotes some Russian text? In the 1980s people began to want to solve this problem, and the Unicode standardization effort began.

Unicode started out using 16-bit characters instead of 8-bit characters. 16 bits means you have $2^{16} = 65,536$ distinct values available, making it possible to represent many different characters from many different alphabets; an initial goal was to have Unicode contain the alphabets for every single human language. It turns out that even 16 bits isn’t enough to meet that goal, and the modern Unicode specification uses a wider range of codes, 0–1,114,111 (0x10ffff in base-16).

There’s a related ISO standard, ISO 10646. Unicode and ISO 10646 were originally separate efforts, but the specifications were merged with the 1.1 revision of Unicode.

(This discussion of Unicode’s history is highly simplified. I don’t think the average Python programmer needs to worry about the historical details; consult the Unicode consortium site listed in the References for more information.)
A **character** is the smallest possible component of a text. ‘A’, ‘B’, ‘C’, etc., are all different characters. So are ‘È’ and ‘Í’. Characters are abstractions, and vary depending on the language or context you’re talking about. For example, the symbol for ohms (Ω) is usually drawn much like the capital letter omega (Ω) in the Greek alphabet (they may even be the same in some fonts), but these are two different characters that have different meanings.

The Unicode standard describes how characters are represented by **code points**. A code point is an integer value, usually denoted in base 16. In the standard, a code point is written using the notation U+12ca to mean the character with value 0x12ca (4810 decimal). The Unicode standard contains a lot of tables listing characters and their corresponding code points:

<table>
<thead>
<tr>
<th>Code Point</th>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0061</td>
<td>'a'</td>
<td>LATIN SMALL LETTER A</td>
</tr>
<tr>
<td>0062</td>
<td>'b'</td>
<td>LATIN SMALL LETTER B</td>
</tr>
<tr>
<td>0063</td>
<td>'c'</td>
<td>LATIN SMALL LETTER C</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>007B</td>
<td>'{'</td>
<td>LEFT CURLY BRACKET</td>
</tr>
</tbody>
</table>

Strictly, these definitions imply that it’s meaningless to say ‘this is character U+12ca’. U+12ca is a code point, which represents some particular character; in this case, it represents the character ‘ETHIOPIAN SYLLABLE WI’. In informal contexts, this distinction between code points and characters will sometimes be forgotten.

A character is represented on a screen or on paper by a set of graphical elements that’s called a **glyph**. The glyph for an uppercase A, for example, is two diagonal strokes and a horizontal stroke, though the exact details will depend on the font being used. Most Python code doesn’t need to worry about glyphs; figuring out the correct glyph to display is generally the job of a GUI toolkit or a terminal’s font renderer.

**Encodings**
To summarize the previous section: a Unicode string is a sequence of code points, which are numbers from 0 to 0x10ffff. This sequence needs to be represented as a set of bytes (meaning, values from 0–255) in memory. The rules for translating a Unicode string into a sequence of bytes are called an encoding.

The first encoding you might think of is an array of 32-bit integers. In this representation, the string “Python” would look like this:

```
P  y  t  h  o
0x50 00 00 00 79 00 00 00 74 00 00 00 68 00 00 00 6f 00 00 00 6
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 2
```

This representation is straightforward but using it presents a number of problems.

1. It’s not portable; different processors order the bytes differently.
2. It’s very wasteful of space. In most texts, the majority of the code points are less than 127, or less than 255, so a lot of space is occupied by zero bytes. The above string takes 24 bytes compared to the 6 bytes needed for an ASCII representation. Increased RAM usage doesn’t matter too much (desktop computers have megabytes of RAM, and strings aren’t usually that large), but expanding our usage of disk and network bandwidth by a factor of 4 is intolerable.
3. It’s not compatible with existing C functions such as strlen(), so a new family of wide string functions would need to be used.
4. Many Internet standards are defined in terms of textual data, and can’t handle content with embedded zero bytes.

Generally people don’t use this encoding, instead choosing other encodings that are more efficient and convenient. UTF-8 is probably the most commonly supported encoding; it will be discussed below.

Encodings don’t have to handle every possible Unicode character,
and most encodings don’t. For example, Python’s default encoding is the ‘ascii’ encoding. The rules for converting a Unicode string into the ASCII encoding are simple; for each code point:

1. If the code point is < 128, each byte is the same as the value of the code point.
2. If the code point is 128 or greater, the Unicode string can’t be represented in this encoding. (Python raises a UnicodeEncodeError exception in this case.)

Latin-1, also known as ISO-8859-1, is a similar encoding. Unicode code points 0–255 are identical to the Latin-1 values, so converting to this encoding simply requires converting code points to byte values; if a code point larger than 255 is encountered, the string can’t be encoded into Latin-1.

Encodings don’t have to be simple one-to-one mappings like Latin-1. Consider IBM’s EBCDIC, which was used on IBM mainframes. Letter values weren’t in one block: ‘a’ through ‘i’ had values from 129 to 137, but ‘j’ through ‘r’ were 145 through 153. If you wanted to use EBCDIC as an encoding, you’d probably use some sort of lookup table to perform the conversion, but this is largely an internal detail.

UTF-8 is one of the most commonly used encodings. UTF stands for “Unicode Transformation Format”, and the ‘8’ means that 8-bit numbers are used in the encoding. (There’s also a UTF-16 encoding, but it’s less frequently used than UTF-8.) UTF-8 uses the following rules:

1. If the code point is <128, it’s represented by the corresponding byte value.
2. If the code point is between 128 and 0x7ff, it’s turned into two byte values between 128 and 255.
3. Code points >0x7ff are turned into three- or four-byte sequences, where each byte of the sequence is between 128
UTF-8 has several convenient properties:

1. It can handle any Unicode code point.
2. A Unicode string is turned into a string of bytes containing no embedded zero bytes. This avoids byte-ordering issues, and means UTF-8 strings can be processed by C functions such as `strcpy()` and sent through protocols that can’t handle zero bytes.
3. A string of ASCII text is also valid UTF-8 text.
4. UTF-8 is fairly compact; the majority of code points are turned into two bytes, and values less than 128 occupy only a single byte.
5. If bytes are corrupted or lost, it’s possible to determine the start of the next UTF-8-encoded code point and resynchronize. It’s also unlikely that random 8-bit data will look like valid UTF-8.

References

The Unicode Consortium site at <http://www.unicode.org> has character charts, a glossary, and PDF versions of the Unicode specification. Be prepared for some difficult reading. <http://www.unicode.org/history/> is a chronology of the origin and development of Unicode.

To help understand the standard, Jukka Korpela has written an introductory guide to reading the Unicode character tables, available at <https://www.cs.tut.fi/~jkorpela/unicode/guide.html>.

Another good introductory article was written by Joel Spolsky <http://www.joelonsoftware.com/articles/Unicode.html>. If this introduction didn’t make things clear to you, you should try reading this alternate article before continuing.
Wikipedia entries are often helpful; see the entries for “character encoding” <http://en.wikipedia.org/wiki/Character_encoding> and UTF-8 <http://en.wikipedia.org/wiki/UTF-8>, for example.
Python 2.x’s Unicode Support

Now that you’ve learned the rudiments of Unicode, we can look at Python’s Unicode features.

The Unicode Type

Unicode strings are expressed as instances of the `unicode` type, one of Python’s repertoire of built-in types. It derives from an abstract type called `basestring`, which is also an ancestor of the `str` type; you can therefore check if a value is a string type with `isinstance(value, basestring)`. Under the hood, Python represents Unicode strings as either 16- or 32-bit integers, depending on how the Python interpreter was compiled.

The `unicode()` constructor has the signature `unicode(string[, encoding, errors])`. All of its arguments should be 8-bit strings. The first argument is converted to Unicode using the specified encoding; if you leave off the `encoding` argument, the ASCII encoding is used for the conversion, so characters greater than 127 will be treated as errors:

```python
>>> unicode('abcdef')
u'abcdef'
>>> s = unicode('abcdef')
>>> type(s)
<type 'unicode'>
>>> unicode('abcdef' + chr(255))
Traceback (most recent call last):
...  UnicodeDecodeError: 'ascii' codec can't decode byte 0xff in position 6:
         ordinal not in range(128)
```

The `errors` argument specifies the response when the input string...
can’t be converted according to the encoding’s rules. Legal values for this argument are ‘strict’ (raise a UnicodeDecodeError exception), ‘replace’ (add U+FFFD, ‘REPLACEMENT CHARACTER’), or ‘ignore’ (just leave the character out of the Unicode result). The following examples show the differences:

```python
>>> unicode(\x80abc, errors='strict')
Traceback (most recent call last):
  ...  
UnicodeDecodeError: 'ascii' codec can't decode byte 0x80 in position 0:
  ordinal not in range(128)
>>> unicode(\x80abc, errors='replace')
u'\ufffdabc'
>>> unicode(\x80abc, errors='ignore')
u'abc'
```

Encodings are specified as strings containing the encoding’s name. Python 2.7 comes with roughly 100 different encodings; see the Python Library Reference at Standard Encodings for a list. Some encodings have multiple names; for example, ‘latin-1’, ‘iso_8859_1’ and ‘8859’ are all synonyms for the same encoding.

One-character Unicode strings can also be created with the unichr() built-in function, which takes integers and returns a Unicode string of length 1 that contains the corresponding code point. The reverse operation is the built-in ord() function that takes a one-character Unicode string and returns the code point value:

```python
>>> unichr(40960)
u'\u0000'
>>> ord(u'\u0000')
40960
```

Instances of the unicode type have many of the same methods as the 8-bit string type for operations such as searching and formatting:

```python
>>> s = u'Was ever feather so lightly blown to and fro as this
```
Note that the arguments to these methods can be Unicode strings or 8-bit strings. 8-bit strings will be converted to Unicode before carrying out the operation; Python's default ASCII encoding will be used, so characters greater than 127 will cause an exception:

```python
>>> s.find('Was\x9f')
Traceback (most recent call last):
 ... 
UnicodeDecodeError: 'ascii' codec can't decode byte 0x9f in position 3: ordinal not in range(128)
```

Much Python code that operates on strings will therefore work with Unicode strings without requiring any changes to the code. (Input and output code needs more updating for Unicode; more on this later.)

Another important method is `.encode([encoding], [errors='strict'])`, which returns an 8-bit string version of the Unicode string, encoded in the requested encoding. The `errors` parameter is the same as the parameter of the `unicode()` constructor, with one additional possibility: as well as ‘strict’, ‘ignore’, and ‘replace’, you can also pass ‘xmlcharrefreplace’ which uses XML's character references. The following example shows the different results:
Python's 8-bit strings have a `.decode([encoding], [errors])` method that interprets the string using the given encoding:

```python
>>> u = unichr(40960) + u'abcd' + unichr(1972)  # Assemble a string
>>> utf8_version = u.encode('utf-8')  # Encode as UTF-8
>>> type(utf8_version), utf8_version
(<type 'str'>, '\xea\x80\x80abcd\xde\xb4')
>>> u2 = utf8_version.decode('utf-8')  # Decode using UTF-8
>>> u == u2  # The two strings match
True
```

The low-level routines for registering and accessing the available encodings are found in the `codecs` module. However, the encoding and decoding functions returned by this module are usually more low-level than is comfortable, so I'm not going to describe the `codecs` module here. If you need to implement a completely new encoding, you'll need to learn about the `codecs` module interfaces, but implementing encodings is a specialized task that also won't be covered here. Consult the Python documentation to learn more about this module.

The most commonly used part of the `codecs` module is the `codecs.open()` function which will be discussed in the section on
Unicode Literals in Python Source Code

In Python source code, Unicode literals are written as strings prefixed with the ‘u’ or ‘U’ character: u'abcdefghijk'. Specific code points can be written using the \u escape sequence, which is followed by four hex digits giving the code point. The \U escape sequence is similar, but expects 8 hex digits, not 4.

Unicode literals can also use the same escape sequences as 8-bit strings, including \x, but \x only takes two hex digits so it can’t express an arbitrary code point. Octal escapes can go up to U+01ff, which is octal 777.

```
>>> s = u"a\xac\u1234\u20ac\U00008000"
... # ^^^^ two-digit hex escape
... # ^^^^^^^ four-digit Unicode escape
... # ^^^^^^^^^ eight-digit Unicode escape
>>> for c in s:  print ord(c),
... 97 172 4660 8364 32768
```

Using escape sequences for code points greater than 127 is fine in small doses, but becomes an annoyance if you’re using many accented characters, as you would in a program with messages in French or some other accent-using language. You can also assemble strings using the unichr() built-in function, but this is even more tedious.

Ideally, you’d want to be able to write literals in your language’s natural encoding. You could then edit Python source code with your favorite editor which would display the accented characters naturally, and have the right characters used at runtime.
Python supports writing Unicode literals in any encoding, but you have to declare the encoding being used. This is done by including a special comment as either the first or second line of the source file:

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-

u = u'abcdé'
print ord(u[-1])
```

The syntax is inspired by Emacs's notation for specifying variables local to a file. Emacs supports many different variables, but Python only supports 'coding'. The ` -*- ` symbols indicate to Emacs that the comment is special; they have no significance to Python but are a convention. Python looks for `coding: name` or `coding=name` in the comment.

If you don’t include such a comment, the default encoding used will be ASCII. Versions of Python before 2.4 were Euro-centric and assumed Latin-1 as a default encoding for string literals; in Python 2.4, characters greater than 127 still work but result in a warning. For example, the following program has no encoding declaration:

```
#!/usr/bin/env python
u = u'abcdé'
print ord(u[-1])
```

When you run it with Python 2.4, it will output the following warning:

```
amk:~$ python2.4 p263.py
sys:1: DeprecationWarning: Non-ASCII character '\xe9'
in file p263.py on line 2, but no encoding declared;
see https://www.python.org/peps/pep-0263.html for details
```

Python 2.5 and higher are stricter and will produce a syntax error:

```
amk:~$ python2.5 p263.py
File "*/tmp/p263.py", line 2
```
Unicode Properties

The Unicode specification includes a database of information about code points. For each code point that’s defined, the information includes the character’s name, its category, the numeric value if applicable (Unicode has characters representing the Roman numerals and fractions such as one-third and four-fifths). There are also properties related to the code point’s use in bidirectional text and other display-related properties.

The following program displays some information about several characters, and prints the numeric value of one particular character:

```python
import unicodedata

u = unichr(233) + unichr(0x0bf2) + unichr(3972) + unichr(6000)

for i, c in enumerate(u):
    print i, '%04x' % ord(c), unicodedata.category(c),
    print unicodedata.name(c)

# Get numeric value of second character
print unicodedata.numeric(u[1])
```

When run, this prints:

```
00e9 Ll LATIN SMALL LETTER E WITH ACUTE
10bf2 No TAMIL NUMBER ONE THOUSAND
20f84 Mn TIBETAN MARK HALANTA
31770 Lo TAGBANWA LETTER SA
433af So SQUARE RAD OVER S SQUARED
1000.0
```

The category codes are abbreviations describing the nature of the character. These are grouped into categories such as “Letter”,

“Number”, “Punctuation”, or “Symbol”, which in turn are broken up into subcategories. To take the codes from the above output, 'Ll' means ‘Letter, lowercase’, 'No' means “Number, other”, 'Mn' is “Mark, nonspacing”, and 'So' is “Symbol, other”. See <http://www.unicode.org/reports/tr44/#General_Category_Values> for a list of category codes.

References

The Unicode and 8-bit string types are described in the Python library reference at Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange.

The documentation for the unicedata module.

The documentation for the codecs module.

Marc-André Lemburg gave a presentation at EuroPython 2002 titled “Python and Unicode”. A PDF version of his slides is available at <https://downloads.egenix.com/python/Unicode-EPC2002-Talk.pdf>, and is an excellent overview of the design of Python’s Unicode features.
Once you’ve written some code that works with Unicode data, the next problem is input/output. How do you get Unicode strings into your program, and how do you convert Unicode into a form suitable for storage or transmission?

It’s possible that you may not need to do anything depending on your input sources and output destinations; you should check whether the libraries used in your application support Unicode natively. XML parsers often return Unicode data, for example. Many relational databases also support Unicode-valued columns and can return Unicode values from an SQL query.

Unicode data is usually converted to a particular encoding before it gets written to disk or sent over a socket. It’s possible to do all the work yourself: open a file, read an 8-bit string from it, and convert the string with \texttt{unicode(str, encoding)}. However, the manual approach is not recommended.

One problem is the multi-byte nature of encodings; one Unicode character can be represented by several bytes. If you want to read the file in arbitrary-sized chunks (say, 1K or 4K), you need to write error-handling code to catch the case where only part of the bytes encoding a single Unicode character are read at the end of a chunk. One solution would be to read the entire file into memory and then perform the decoding, but that prevents you from working with files that are extremely large; if you need to read a 2Gb file, you need 2Gb of RAM. (More, really, since for at least a moment you’d need to have both the encoded string and its Unicode version in memory.)

The solution would be to use the low-level decoding interface to catch the case of partial coding sequences. The work of implementing this has already been done for you: the \texttt{codecs} module
includes a version of the `open()` function that returns a file-like object that assumes the file's contents are in a specified encoding and accepts Unicode parameters for methods such as `.read()` and `.write()`.

The function's parameters are `open(filename, mode='rb', encoding=None, errors='strict', buffering=1)`. `mode` can be `'r'`, `'w'`, or `'a'`, just like the corresponding parameter to the regular built-in `open()` function; add a '+' to update the file. `buffering` is similarly parallel to the standard function's parameter. `encoding` is a string giving the encoding to use; if it's left as `None`, a regular Python file object that accepts 8-bit strings is returned. Otherwise, a wrapper object is returned, and data written to or read from the wrapper object will be converted as needed. `errors` specifies the action for encoding errors and can be one of the usual values of ‘strict’, ‘ignore’, and ‘replace’.

Reading Unicode from a file is therefore simple:

```python
import codecs
f = codecs.open('unicode.rst', encoding='utf-8')
for line in f:
    print repr(line)
```

It's also possible to open files in update mode, allowing both reading and writing:

```python
f = codecs.open('test', encoding='utf-8', mode='w+')
f.write(u'\u4500 blah blah blah
')
f.seek(0)
print repr(f.readline()[:1])
f.close()
```

Unicode character U+FEFF is used as a byte-order mark (BOM), and is often written as the first character of a file in order to assist with autodetection of the file's byte ordering. Some encodings, such
as UTF-16, expect a BOM to be present at the start of a file; when such an encoding is used, the BOM will be automatically written as the first character and will be silently dropped when the file is read. There are variants of these encodings, such as ‘utf-16-le' and 'utf-16-be' for little-endian and big-endian encodings, that specify one particular byte ordering and don’t skip the BOM.

### Unicode filenames

Most of the operating systems in common use today support filenames that contain arbitrary Unicode characters. Usually this is implemented by converting the Unicode string into some encoding that varies depending on the system. For example, Mac OS X uses UTF-8 while Windows uses a configurable encoding; on Windows, Python uses the name “mbcs” to refer to whatever the currently configured encoding is. On Unix systems, there will only be a filesystem encoding if you’ve set the `LANG` or `LC_CTYPE` environment variables; if you haven’t, the default encoding is ASCII.

The `sys.getfilesystemencoding()` function returns the encoding to use on your current system, in case you want to do the encoding manually, but there’s not much reason to bother. When opening a file for reading or writing, you can usually just provide the Unicode string as the filename, and it will be automatically converted to the right encoding for you:

```python
filename = u'filename\u4500abc'
f = open(filename, 'w')
f.write('blah\n')
f.close()
```

Functions in the `os` module such as `os.stat()` will also accept Unicode filenames.

`os.listdir()`, which returns filenames, raises an issue: should it
return the Unicode version of filenames, or should it return 8-bit strings containing the encoded versions?\texttt{os.listdir()} will do both, depending on whether you provided the directory path as an 8-bit string or a Unicode string. If you pass a Unicode string as the path, filenames will be decoded using the filesystem's encoding and a list of Unicode strings will be returned, while passing an 8-bit path will return the 8-bit versions of the filenames. For example, assuming the default filesystem encoding is UTF-8, running the following program:

```python
fn = u'filename\u4500abc'
f = open(fn, 'w')
f.close()

import os
print os.listdir('.')
print os.listdir(u'.')
```

will produce the following output:

```
amk:~$ python t.py
['.svn', 'filename\xe4\x94\x80abc', ...]
[u'.svn', u'filename\u4500abc', ...]
```

The first list contains UTF-8-encoded filenames, and the second list contains the Unicode versions.

**Tips for Writing Unicode-aware Programs**

This section provides some suggestions on writing software that deals with Unicode.

The most important tip is:

Software should only work with Unicode strings internally, converting to a particular encoding on output.

If you attempt to write processing functions that accept both Unicode
and 8-bit strings, you will find your program vulnerable to bugs wherever you combine the two different kinds of strings. Python's default encoding is ASCII, so whenever a character with an ASCII value > 127 is in the input data, you'll get a `UnicodeDecodeError` because that character can't be handled by the ASCII encoding.

It's easy to miss such problems if you only test your software with data that doesn't contain any accents; everything will seem to work, but there's actually a bug in your program waiting for the first user who attempts to use characters > 127. A second tip, therefore, is:

Include characters > 127 and, even better, characters > 255 in your test data.

When using data coming from a web browser or some other untrusted source, a common technique is to check for illegal characters in a string before using the string in a generated command line or storing it in a database. If you're doing this, be careful to check the string once it's in the form that will be used or stored; it's possible for encodings to be used to disguise characters. This is especially true if the input data also specifies the encoding; many encodings leave the commonly checked-for characters alone, but Python includes some encodings such as `'base64'` that modify every single character.

For example, let's say you have a content management system that takes a Unicode filename, and you want to disallow paths with a '/' character. You might write this code:

```python
def read_file (filename, encoding):
    if '/' in filename:
        raise ValueError("'/' not allowed in filenames")
    unicode_name = filename.decode(encoding)
f = open(unicode_name, 'r')
# ... return contents of file ...
```
However, if an attacker could specify the 'base64' encoding, they could pass 'L2V0Yy9wYXNzd2Q=', which is the base-64 encoded form of the string '/etc/passwd', to read a system file. The above code looks for '/' characters in the encoded form and misses the dangerous character in the resulting decoded form.

References

The PDF slides for Marc-André Lemburg's presentation “Writing Unicode-aware Applications in Python” are available at <https://downloads.egenix.com/python/LSM2005-Developing-Unicode-aware-applications-in-Python.pdf> and discuss questions of character encodings as well as how to internationalize and localize an application.
Revision History and Acknowledgements

Thanks to the following people who have noted errors or offered suggestions on this article: Nicholas Bastin, Marius Gedminas, Kent Johnson, Ken Krugler, Marc-André Lemburg, Martin von Löwis, Chad Whitacre.

Version 1.0: posted August 5 2005.

Version 1.01: posted August 7 2005. Corrects factual and markup errors; adds several links.


Version 1.03: posted June 20 2010. Notes that Python 3.x is not covered, and that the HOWTO only covers 2.x.
HOWTO Fetch Internet Resources Using urllib2

Author: Michael Foord

Note: There is a French translation of an earlier revision of this HOWTO, available at urllib2 - Le Manuel manquant.
**Introduction**

**urllib2** is a Python module for fetching URLs (Uniform Resource Locators). It offers a very simple interface, in the form of the `urlopen` function. This is capable of fetching URLs using a variety of different protocols. It also offers a slightly more complex interface for handling common situations - like basic authentication, cookies, proxies and so on. These are provided by objects called handlers and openers.

`urllib2` supports fetching URLs for many “URL schemes” (identified by the string before the “:” in URL - for example “ftp” is the URL scheme of “ftp://python.org/”) using their associated network protocols (e.g. FTP, HTTP). This tutorial focuses on the most common case, HTTP.

For straightforward situations `urlopen` is very easy to use. But as soon as you encounter errors or non-trivial cases when opening HTTP URLs, you will need some understanding of the HyperText Transfer Protocol. The most comprehensive and authoritative reference to HTTP is RFC 2616. This is a technical document and not intended to be easy to read. This HOWTO aims to illustrate using `urllib2`, with enough detail about HTTP to help you through. It is not intended to replace the `urllib2` docs, but is supplementary to them.

---

**Related Articles**

You may also find useful the following article on fetching web resources with Python:

- **Basic Authentication**

  A tutorial on *Basic Authentication*, with examples in Python.
Fetching URLs

The simplest way to use urllib2 is as follows:

```python
import urllib2
response = urllib2.urlopen('http://python.org/)
html = response.read()
```

Many uses of urllib2 will be that simple (note that instead of an ‘http:’ URL we could have used a URL starting with ‘ftp:’, ‘file:’, etc.). However, it’s the purpose of this tutorial to explain the more complicated cases, concentrating on HTTP.

HTTP is based on requests and responses - the client makes requests and servers send responses. urllib2 mirrors this with a Request object which represents the HTTP request you are making. In its simplest form you create a Request object that specifies the URL you want to fetch. Calling `urlopen` with this Request object returns a response object for the URL requested. This response is a file-like object, which means you can for example call `read()` on the response:

```python
import urllib2
req = urllib2.Request('http://www.voidspace.org.uk')
response = urllib2.urlopen(req)
the_page = response.read()
```

Note that urllib2 makes use of the same Request interface to handle all URL schemes. For example, you can make an FTP request like so:

```python
req = urllib2.Request('ftp://example.com/')
```

In the case of HTTP, there are two extra things that Request objects
allow you to do: First, you can pass data to be sent to the server. Second, you can pass extra information (“metadata”) about the data or the about request itself, to the server - this information is sent as HTTP “headers”. Let's look at each of these in turn.

## Data

Sometimes you want to send data to a URL (often the URL will refer to a CGI (Common Gateway Interface) script [1] or other web application). With HTTP, this is often done using what's known as a POST request. This is often what your browser does when you submit a HTML form that you filled in on the web. Not all POSTs have to come from forms: you can use a POST to transmit arbitrary data to your own application. In the common case of HTML forms, the data needs to be encoded in a standard way, and then passed to the Request object as the data argument. The encoding is done using a function from the urllib library not from urllib2.

```python
import urllib
import urllib2

url = 'http://www.someserver.com/cgi-bin/register.cgi'
values = {'name' : 'Michael Foord',
          'location' : 'Northampton',
          'language' : 'Python' }

data = urllib.urlencode(values)
req = urllib2.Request(url, data)
response = urllib2.urlopen(req)
the_page = response.read()
```

Note that other encodings are sometimes required (e.g. for file upload from HTML forms - see HTML Specification, Form Submission for more details).

If you do not pass the data argument, urllib2 uses a GET request. One way in which GET and POST requests differ is that POST
requests often have “side-effects”: they change the state of the system in some way (for example by placing an order with the website for a hundredweight of tinned spam to be delivered to your door). Though the HTTP standard makes it clear that POSTs are intended to always cause side-effects, and GET requests never to cause side-effects, nothing prevents a GET request from having side-effects, nor a POST requests from having no side-effects. Data can also be passed in an HTTP GET request by encoding it in the URL itself.

This is done as follows:

```python
>>> import urllib2
>>> import urllib
>>> data = {}
>>> data['name'] = 'Somebody Here'
>>> data['location'] = 'Northampton'
>>> data['language'] = 'Python'
>>> url_values = urlliburlencode(data)
>>> print url_values  # The order may differ.
name=Somebody+Here&language=Python&location=Northampton
>>> url = 'http://www.example.com/example.cgi'
>>> full_url = url + '?' + url_values
>>> data = urllib2.urlopen(full_url)
```

Notice that the full URL is created by adding a `?` to the URL, followed by the encoded values.

### Headers

We'll discuss here one particular HTTP header, to illustrate how to add headers to your HTTP request.

Some websites [2] dislike being browsed by programs, or send different versions to different browsers [3]. By default urllib2 identifies itself as `Python-urllib/x.y` (where `x` and `y` are the major and minor version numbers of the Python release, e.g. `Python-urllib/2.5`),
which may confuse the site, or just plain not work. The way a browser identifies itself is through the `User-Agent` header [4]. When you create a Request object you can pass a dictionary of headers in. The following example makes the same request as above, but identifies itself as a version of Internet Explorer [5].

```python
import urllib
import urllib2

url = 'http://www.someserver.com/cgi-bin/register.cgi'
user_agent = 'Mozilla/5.0 (Windows NT 6.1; Win64; x64)'
values = {'name': 'Michael Foord',
          'location': 'Northampton',
          'language': 'Python'}
headers = {'User-Agent': user_agent}

data = urllib.urlencode(values)
req = urllib2.Request(url, data, headers)
response = urllib2.urlopen(req)
the_page = response.read()
```

The response also has two useful methods. See the section on `info` and `geturl` which comes after we have a look at what happens when things go wrong.
Handling Exceptions

`urlopen` raises `URLError` when it cannot handle a response (though as usual with Python APIs, built-in exceptions such as `ValueError`, `TypeError` etc. may also be raised).

`HTTPError` is the subclass of `URLError` raised in the specific case of HTTP URLs.

**URLError**

Often, `URLError` is raised because there is no network connection (no route to the specified server), or the specified server doesn't exist. In this case, the exception raised will have a 'reason' attribute, which is a tuple containing an error code and a text error message.

e.g.

```python
>>> req = urllib2.Request('http://www.pretend_server.org')
>>> try: urllib2.urlopen(req)
... except URLError as e:
...     print e.reason
...
(4, 'getaddrinfo failed')
```

**HTTPError**

Every HTTP response from the server contains a numeric “status code”. Sometimes the status code indicates that the server is unable to fulfil the request. The default handlers will handle some of these responses for you (for example, if the response is a “redirection” that requests the client fetch the document from a different URL, `urllib2` will handle that for you). For those it can’t handle, `urlopen` will raise an `HTTPError`. Typical errors include ‘404’ (page not found), ‘403’
(request forbidden), and ‘401’ (authentication required).

See section 10 of RFC 2616 for a reference on all the HTTP error codes.

The `HTTPError` instance raised will have an integer ‘code’ attribute, which corresponds to the error sent by the server.

## Error Codes

Because the default handlers handle redirects (codes in the 300 range), and codes in the 100–299 range indicate success, you will usually only see error codes in the 400–599 range.

`BaseHTTPServer.BaseHTTPRequestHandler.responses` is a useful dictionary of response codes in that shows all the response codes used by RFC 2616. The dictionary is reproduced here for convenience.

```python
# Table mapping response codes to messages; entries have the
# form {code: (shortmessage, longmessage)}.
responses = {
    100: ('Continue', 'Request received, please continue'),
    101: ('Switching Protocols', 'Switching to new protocol; obey Upgrade header'),
    200: ('OK', 'Request fulfilled, document follows'),
    201: ('Created', 'Document created, URL follows'),
    202: ('Accepted', 'Request accepted, processing continues off-line'),
    203: ('Non-Authoritative Information', 'Request fulfilled from cache'),
    204: ('No Content', 'Request fulfilled, nothing follows'),
    205: ('Reset Content', 'Clear input form for further input'),
    206: ('Partial Content', 'Partial content follows'),
    300: ('Multiple Choices', 'Object has several resources -- see URI list'),
    301: ('Moved Permanently', 'Object moved permanently -- see URI list'),
    302: ('Found', 'Object moved temporarily -- see URI list'),
    303: ('See Other', 'Object moved -- see Method and URL list'),
    304: ('Not Modified',
```
{'Document has not changed since given time'),
305: ('Use Proxy',
   'You must use proxy specified in Location to access this resource'),
307: ('Temporary Redirect',
   'Object moved temporarily -- see URI list'),
400: ('Bad Request',
   'Bad request syntax or unsupported method'),
401: ('Unauthorized',
   'No permission -- see authorization schemes'),
402: ('Payment Required',
   'No payment -- see charging schemes'),
403: ('Forbidden',
   'Request forbidden -- authorization will not help'),
404: ('Not Found',
   'Nothing matches the given URI'),
405: ('Method Not Allowed',
   'Specified method is invalid for this server.'),
406: ('Not Acceptable',
   'URI not available in preferred format'),
407: ('Proxy Authentication Required',
   'You must authenticate with this proxy before proceeding.'),
408: ('Request Timeout',
   'Request timed out; try again later'),
409: ('Conflict',
   'Request conflict'),
410: ('Gone',
   'URI no longer exists and has been permanently removed'),
411: ('Length Required',
   'Client must specify Content-Length'),
412: ('Precondition Failed',
   'Precondition in headers is false'),
413: ('Request Entity Too Large',
   'Entity is too large.'),
414: ('Request-URI Too Long',
   'URI is too long.'),
415: ('Unsupported Media Type',
   'Entity body in unsupported format'),
416: ('Requested Range Not Satisfiable',
   'Cannot satisfy request range.'),
417: ('Expectation Failed',
   'Expect condition could not be satisfied.'),
500: ('Internal Server Error',
   'Server got itself in trouble'),
501: ('Not Implemented',
   'Server does not support this operation'),
502: ('Bad Gateway',
   'Invalid responses from another server'),
503: ('Service Unavailable',
   'The server cannot process the request due to a high'),
504: ('Gateway Timeout',
   'The gateway server did not receive a timely response'),
505: ('HTTP Version Not Supported',
   'Cannot fulfill request')
When an error is raised the server responds by returning an HTTP error code and an error page. You can use the `HTTPError` instance as a response on the page returned. This means that as well as the code attribute, it also has read, geturl, and info, methods.

```python
>>> req = urllib2.Request('http://www.python.org/fish.html')
>>> try:
...     urllib2.urlopen(req)
...     except urllib2.HTTPError as e:
...         print e.code
...         print e.read()
...     404
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
...
<title>Page Not Found</title>
...
```

**Wrapping it Up**

So if you want to be prepared for `HTTPError` or `URLError` there are two basic approaches. I prefer the second approach.

### Number 1

```python
from urllib2 import Request, urlopen, URLError, HTTPError

req = Request(someurl)
try:
    response = urlopen(req)
except HTTPError as e:
    print 'The server couldn\'t fulfill the request.'
    print 'Error code: ', e.code
except URLError as e:
    print 'We failed to reach a server.'
    print 'Reason: ', e.reason
else:
    # everything is fine
```

**Note:** The `except HTTPError` must come first, otherwise `except`
URLError will also catch an HTTPError.

Number 2

```python
from urllib2 import Request, urlopen, URLError
req = Request(someurl)
try:
    response = urlopen(req)
except URLError as e:
    if hasattr(e, 'reason'):
        print 'We failed to reach a server.'
        print 'Reason: ', e.reason
    elif hasattr(e, 'code'):
        print 'The server couldn\'t fulfill the request.'
        print 'Error code: ', e.code
else:
    # everything is fine
```
info and geturl

The response returned by urlopen (or the HTTPError instance) has two useful methods info() and geturl().

**geturl** - this returns the real URL of the page fetched. This is useful because urlopen (or the opener object used) may have followed a redirect. The URL of the page fetched may not be the same as the URL requested.

**info** - this returns a dictionary-like object that describes the page fetched, particularly the headers sent by the server. It is currently an httpclient.HTTPMessage instance.

Typical headers include ‘Content-length’, ‘Content-type’, and so on. See the Quick Reference to HTTP Headers for a useful listing of HTTP headers with brief explanations of their meaning and use.
Openers and Handlers

When you fetch a URL you use an opener (an instance of the perhaps confusingly-named `urllib2.OpenerDirector`). Normally we have been using the default opener - via `urlopen` - but you can create custom openers. Openers use handlers. All the “heavy lifting” is done by the handlers. Each handler knows how to open URLs for a particular URL scheme (http, ftp, etc.), or how to handle an aspect of URL opening, for example HTTP redirec\[d\]ions or HTTP cookies.

You will want to create openers if you want to fetch URLs with specific handlers installed, for example to get an opener that handles cookies, or to get an opener that does not handle redirec\[d\]ions.

To create an opener, instantiate an `OpenerDirector`, and then call `add_handler(some_handler_instance)` repeatedly.

Alternatively, you can use `build_opener`, which is a convenience function for creating opener objects with a single function call. `build_opener` add\[s\] several handlers by default, but provides a quick way to add more and/or override the default handlers.

Other sorts of handlers you might want to can handle proxies, authentication, and other common but slightly specialised situations.

`install_opener` can be used to make an `opener` object the (global) default opener. This means that calls to `urlopen` will use the opener you have installed.

Opener objects have an `open` method, which can be called directly to fetch urls in the same way as the `urlopen` function: there's no need to call `install_opener`, except as a convenience.
Basic Authentication

To illustrate creating and installing a handler we will use the HTTPBasicAuthHandler. For a more detailed discussion of this subject – including an explanation of how Basic Authentication works - see the Basic Authentication Tutorial.

When authentication is required, the server sends a header (as well as the 401 error code) requesting authentication. This specifies the authentication scheme and a ‘realm’. The header looks like: \texttt{WWW-Authenticate: SCHEME realm="REALM"}.

\texttt{WWW-Authenticate: Basic realm="cPanel Users"}

The client should then retry the request with the appropriate name and password for the realm included as a header in the request. This is ‘basic authentication’. In order to simplify this process we can create an instance of \texttt{HTTPBasicAuthHandler} and an opener to use this handler.

The \texttt{HTTPBasicAuthHandler} uses an object called a password manager to handle the mapping of URLs and realms to passwords and usernames. If you know what the realm is (from the authentication header sent by the server), then you can use a \texttt{HTTPPasswordMgr}. Frequently one doesn’t care what the realm is. In that case, it is convenient to use \texttt{HTTPPasswordMgrWithDefaultRealm}. This allows you to specify a default username and password for a URL. This will be supplied in the absence of you providing an alternative combination for a specific realm. We indicate this by providing \texttt{None} as the realm argument to the \texttt{add_password} method.
The top-level URL is the first URL that requires authentication. URLs "deeper" than the URL you pass to .add_password() will also match.

```python
# create a password manager
password_mgr = urllib2.HTTPPasswordMgrWithDefaultRealm()

# Add the username and password.
# If we knew the realm, we could use it instead of None.
top_level_url = "http://example.com/foo/
password_mgr.add_password(None, top_level_url, username, passwo
d
handler = urllib2.HTTPBasicAuthHandler(password_mgr)

# create "opener" (OpenerDirector instance)
 opener = urllib2.build_opener(handler)

# use the opener to fetch a URL
 opener.open(a_url)

# Install the opener.
# Now all calls to urllib2.urlopen use our opener.
 urllib2.install_opener(opener)
```

**Note:** In the above example we only supplied our HTTPBasicAuthHandler to build_opener. By default openers have the handlers for normal situations – ProxyHandler (if a proxy setting such as an http_proxy environment variable is set), UnknownHandler, HTTPHandler, HTTPDefaultErrorHandler, HTTPRedirectHandler, FTPHandler, FileHandler, HTTPErrorProcessor.

top_level_url is in fact either a full URL (including the 'http:' scheme component and the hostname and optionally the port number) e.g. “http://example.com/” or an “authority” (i.e. the hostname, optionally including the port number) e.g. “example.com” or “example.com:8080” (the latter example includes a port number). The authority, if present, must NOT contain the “userinfo” component - for example “joe:password@example.com” is not correct.
Proxies

urllib2 will auto-detect your proxy settings and use those. This is through the ProxyHandler, which is part of the normal handler chain when a proxy setting is detected. Normally that’s a good thing, but there are occasions when it may not be helpful [6]. One way to do this is to setup our own ProxyHandler, with no proxies defined. This is done using similar steps to setting up a Basic Authentication handler:

```python
>>> proxy_support = urllib2.ProxyHandler({})
>>> opener = urllib2.build_opener(proxy_support)
>>> urllib2.install_opener(opener)
```

**Note:** Currently urllib2 does not support fetching of https locations through a proxy. However, this can be enabled by extending urllib2 as shown in the recipe [7].

**Note:** HTTP_PROXY will be ignored if a variable REQUEST_METHOD is set; see the documentation on getproxies().
Sockets and Layers

The Python support for fetching resources from the web is layered. urllib2 uses the httpplib library, which in turn uses the socket library.

As of Python 2.3 you can specify how long a socket should wait for a response before timing out. This can be useful in applications which have to fetch web pages. By default the socket module has no timeout and can hang. Currently, the socket timeout is not exposed at the httpplib or urllib2 levels. However, you can set the default timeout globally for all sockets using

```python
import socket
import urllib2

# timeout in seconds
timeout = 10
socket.setdefaulttimeout(timeout)

# this call to urllib2.urlopen now uses the default timeout
# we have set in the socket module
req = urllib2.Request('http://www.voidspace.org.uk')
response = urllib2.urlopen(req)
```
This document was reviewed and revised by John Lee.

[1] For an introduction to the CGI protocol see Writing Web Applications in Python.


Browser sniffing is a very bad practice for website design - building sites using web standards is much more sensible. Unfortunately a lot of sites still send different versions to different browsers.

[3] The user agent for MSIE 6 is ‘Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)’

[4] For details of more HTTP request headers, see Quick Reference to HTTP Headers.

In my case I have to use a proxy to access the internet at work. If you attempt to fetch localhost URLs through this proxy it blocks them. IE is set to use the proxy, which urllib2 picks up on. In order to test scripts with a localhost server, I have to prevent urllib2 from using the proxy.

HOWTO Use Python in the web

Author: Marek Kubica

Abstract

This document shows how Python fits into the web. It presents some ways to integrate Python with a web server, and general practices useful for developing web sites.

Programming for the Web has become a hot topic since the rise of “Web 2.0”, which focuses on user-generated content on web sites. It has always been possible to use Python for creating web sites, but it was a rather tedious task. Therefore, many frameworks and helper tools have been created to assist developers in creating faster and more robust sites. This HOWTO describes some of the methods used to combine Python with a web server to create dynamic content. It is not meant as a complete introduction, as this topic is far too broad to be covered in one single document. However, a short overview of the most popular libraries is provided.

See also: While this HOWTO tries to give an overview of Python in the web, it cannot always be as up to date as desired. Web development in Python is rapidly moving forward, so the wiki page on Web Programming may be more in sync with recent development.
The Low-Level View

When a user enters a web site, their browser makes a connection to the site’s web server (this is called the request). The server looks up the file in the file system and sends it back to the user’s browser, which displays it (this is the response). This is roughly how the underlying protocol, HTTP, works.

Dynamic web sites are not based on files in the file system, but rather on programs which are run by the web server when a request comes in, and which generate the content that is returned to the user. They can do all sorts of useful things, like display the postings of a bulletin board, show your email, configure software, or just display the current time. These programs can be written in any programming language the server supports. Since most servers support Python, it is easy to use Python to create dynamic web sites.

Most HTTP servers are written in C or C++, so they cannot execute Python code directly – a bridge is needed between the server and the program. These bridges, or rather interfaces, define how programs interact with the server. There have been numerous attempts to create the best possible interface, but there are only a few worth mentioning.

Not every web server supports every interface. Many web servers only support old, now-obsolete interfaces; however, they can often be extended using third-party modules to support newer ones.

Common Gateway Interface

This interface, most commonly referred to as “CGI”, is the oldest, and is supported by nearly every web server out of the box. Programs using CGI to communicate with their web server need to
be started by the server for every request. So, every request starts a new Python interpreter – which takes some time to start up – thus making the whole interface only usable for low load situations.

The upside of CGI is that it is simple – writing a Python program which uses CGI is a matter of about three lines of code. This simplicity comes at a price: it does very few things to help the developer.

Writing CGI programs, while still possible, is no longer recommended. With WSGI, a topic covered later in this document, it is possible to write programs that emulate CGI, so they can be run as CGI if no better option is available.

See also: The Python standard library includes some modules that are helpful for creating plain CGI programs:

- **cgi** – Handling of user input in CGI scripts
- **cgitb** – Displays nice tracebacks when errors happen in CGI applications, instead of presenting a “500 Internal Server Error” message

The Python wiki features a page on CGI scripts with some additional information about CGI in Python.

Simple script for testing CGI

To test whether your web server works with CGI, you can use this short and simple CGI program:

```bash
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

# enable debugging
import cgitb
cgitb.enable()
```
print "Content-Type: text/plain; charset=utf-8"
print
print "Hello World!"

Depending on your web server configuration, you may need to save this code with a .py or .cgi extension. Additionally, this file may also need to be in a cgi-bin folder, for security reasons.

You might wonder what the cgitb line is about. This line makes it possible to display a nice traceback instead of just crashing and displaying an “Internal Server Error” in the user’s browser. This is useful for debugging, but it might risk exposing some confidential data to the user. You should not use cgitb in production code for this reason. You should always catch exceptions, and display proper error pages – end-users don’t like to see nondescript “Internal Server Errors” in their browsers.

### Setting up CGI on your own server

If you don’t have your own web server, this does not apply to you. You can check whether it works as-is, and if not you will need to talk to the administrator of your web server. If it is a big host, you can try filing a ticket asking for Python support.

If you are your own administrator or want to set up CGI for testing purposes on your own computers, you have to configure it by yourself. There is no single way to configure CGI, as there are many web servers with different configuration options. Currently the most widely used free web server is Apache HTTPd, or Apache for short. Apache can be easily installed on nearly every system using the system’s package management tool. lighttpd is another alternative and is said to have better performance. On many systems this server can also be installed using the package management tool, so manually compiling the web server may not be needed.
On Apache you can take a look at the Dynamic Content with CGI tutorial, where everything is described. Most of the time it is enough just to set +ExecCGI. The tutorial also describes the most common gotchas that might arise.

On lighttpd you need to use the CGI module, which can be configured in a straightforward way. It boils down to setting cgi.assign properly.

Common problems with CGI scripts

Using CGI sometimes leads to small annoyances while trying to get these scripts to run. Sometimes a seemingly correct script does not work as expected, the cause being some small hidden problem that’s difficult to spot.

Some of these potential problems are:

- The Python script is not marked as executable. When CGI scripts are not executable most web servers will let the user download it, instead of running it and sending the output to the user. For CGI scripts to run properly on Unix-like operating systems, the +x bit needs to be set. Using chmod a+x your_script.py may solve this problem.

- On a Unix-like system, The line endings in the program file must be Unix style line endings. This is important because the web server checks the first line of the script (called shebang) and tries to run the program specified there. It gets easily confused by Windows line endings (Carriage Return & Line Feed, also called CRLF), so you have to convert the file to Unix line endings (only Line Feed, LF). This can be done automatically by uploading the file via FTP in text mode instead of binary mode, but the preferred way is just telling your editor to save the files with Unix line endings. Most editors support this.

- Your web server must be able to read the file, and you need to
make sure the permissions are correct. On unix-like systems, the server often runs as user and group `www-data`, so it might be worth a try to change the file ownership, or making the file world readable by using `chmod a+r your_script.py`.

- The web server must know that the file you're trying to access is a CGI script. Check the configuration of your web server, as it may be configured to expect a specific file extension for CGI scripts.
- On Unix-like systems, the path to the interpreter in the shebang (`#!/usr/bin/env python`) must be correct. This line calls `/usr/bin/env` to find Python, but it will fail if there is no `/usr/bin/env`, or if Python is not in the web server's path. If you know where your Python is installed, you can also use that full path. The commands `whereis python` and `type -p python` could help you find where it is installed. Once you know the path, you can change the shebang accordingly: `#!/usr/bin/python`.
- The file must not contain a BOM (Byte Order Mark). The BOM is meant for determining the byte order of UTF-16 and UTF-32 encodings, but some editors write this also into UTF-8 files. The BOM interferes with the shebang line, so be sure to tell your editor not to write the BOM.
- If the web server is using `mod_python`, `mod_python` may be having problems. `mod_python` is able to handle CGI scripts by itself, but it can also be a source of issues.

mod_python

People coming from PHP often find it hard to grasp how to use Python in the web. Their first thought is mostly `mod_python`, because they think that this is the equivalent to `mod_php`. Actually, there are many differences. What `mod_python` does is embed the interpreter into the Apache process, thus speeding up requests by not having to start a Python interpreter for each request. On the
other hand, it is not “Python intermixed with HTML” in the way that PHP is often intermixed with HTML. The Python equivalent of that is a template engine. ```mod_python``` itself is much more powerful and provides more access to Apache internals. It can emulate CGI, work in a “Python Server Pages” mode (similar to JSP) which is “HTML intermingled with Python”, and it has a “Publisher” which designates one file to accept all requests and decide what to do with them.

```mod_python``` does have some problems. Unlike the PHP interpreter, the Python interpreter uses caching when executing files, so changes to a file will require the web server to be restarted. Another problem is the basic concept – Apache starts child processes to handle the requests, and unfortunately every child process needs to load the whole Python interpreter even if it does not use it. This makes the whole web server slower. Another problem is that, because ```mod_python``` is linked against a specific version of ```libpython```, it is not possible to switch from an older version to a newer (e.g. 2.4 to 2.5) without recompiling ```mod_python```. ```mod_python``` is also bound to the Apache web server, so programs written for ```mod_python``` cannot easily run on other web servers.

These are the reasons why ```mod_python``` should be avoided when writing new programs. In some circumstances it still might be a good idea to use ```mod_python``` for deployment, but WSGI makes it possible to run WSGI programs under ```mod_python``` as well.

### FastCGI and SCGI

FastCGI and SCGI try to solve the performance problem of CGI in another way. Instead of embedding the interpreter into the web server, they create long-running background processes. There is still a module in the web server which makes it possible for the web server to “speak” with the background process. As the background process is independent of the server, it can be written in any
language, including Python. The language just needs to have a library which handles the communication with the webserver.

The difference between FastCGI and SCGI is very small, as SCGI is essentially just a “simpler FastCGI”. As the web server support for SCGI is limited, most people use FastCGI instead, which works the same way. Almost everything that applies to SCGI also applies to FastCGI as well, so we'll only cover the latter.

These days, FastCGI is never used directly. Just like mod_python, it is only used for the deployment of WSGI applications.

### Setting up FastCGI

Each web server requires a specific module.

- Apache has both mod_fastcgi and mod_fcgid. mod_fastcgi is the original one, but it has some licensing issues, which is why it is sometimes considered non-free. mod_fcgid is a smaller, compatible alternative. One of these modules needs to be loaded by Apache.
- lighttpd ships its own FastCGI module as well as an SCGI module.
- nginx also supports FastCGI.

Once you have installed and configured the module, you can test it with the following WSGI-application:

```python
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

from cgi import escape
import sys, os
from flup.server.fcgi import WSGIServer

def app(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
```
mod_wsgi

mod_wsgi is an attempt to get rid of the low level gateways. Given that FastCGI, SCGI, and mod_python are mostly used to deploy WSGI applications, mod_wsgi was started to directly embed WSGI applications into the Apache web server. mod_wsgi is specifically designed to host WSGI applications. It makes the deployment of WSGI applications much easier than deployment using other low level methods, which need glue code. The downside is that mod_wsgi is limited to the Apache web server; other servers would need their own implementations of mod_wsgi.

mod_wsgi supports two modes: embedded mode, in which it integrates with the Apache process, and daemon mode, which is more FastCGI-like. Unlike FastCGI, mod_wsgi handles the worker-processes by itself, which makes administration easier.
Step back: WSGI

WSGI has already been mentioned several times, so it has to be something important. In fact it really is, and now it is time to explain it.

The Web Server Gateway Interface, or WSGI for short, is defined in PEP 333 and is currently the best way to do Python web programming. While it is great for programmers writing frameworks, a normal web developer does not need to get in direct contact with it. When choosing a framework for web development it is a good idea to choose one which supports WSGI.

The big benefit of WSGI is the unification of the application programming interface. When your program is compatible with WSGI – which at the outer level means that the framework you are using has support for WSGI – your program can be deployed via any web server interface for which there are WSGI wrappers. You do not need to care about whether the application user uses mod_python or FastCGI or mod_wsgi – with WSGI your application will work on any gateway interface. The Python standard library contains its own WSGI server, wsgiref, which is a small web server that can be used for testing.

A really great WSGI feature is middleware. Middleware is a layer around your program which can add various functionality to it. There is quite a bit of middleware already available. For example, instead of writing your own session management (HTTP is a stateless protocol, so to associate multiple HTTP requests with a single user your application must create and manage such state via a session), you can just download middleware which does that, plug it in, and get on with coding the unique parts of your application. The same thing with compression – there is existing middleware which handles
compressing your HTML using gzip to save on your server’s bandwidth. Authentication is another problem that is easily solved using existing middleware.

Although WSGI may seem complex, the initial phase of learning can be very rewarding because WSGI and the associated middleware already have solutions to many problems that might arise while developing web sites.

WSGI Servers

The code that is used to connect to various low level gateways like CGI or mod_python is called a WSGI server. One of these servers is flup, which supports FastCGI and SCGI, as well as AJP. Some of these servers are written in Python, as flup is, but there also exist others which are written in C and can be used as drop-in replacements.

There are many servers already available, so a Python web application can be deployed nearly anywhere. This is one big advantage that Python has compared with other web technologies.

See also: A good overview of WSGI-related code can be found in the WSGI homepage, which contains an extensive list of WSGI servers which can be used by any application supporting WSGI.

You might be interested in some WSGI-supporting modules already contained in the standard library, namely:

- wsgiref – some tiny utilities and servers for WSGI

Case study: MoinMoin

What does WSGI give the web application developer? Let's take a
look at an application that’s been around for a while, which was written in Python without using WSGI.

One of the most widely used wiki software packages is MoinMoin. It was created in 2000, so it predates WSGI by about three years. Older versions needed separate code to run on CGI, mod_python, FastCGI and standalone.

It now includes support for WSGI. Using WSGI, it is possible to deploy MoinMoin on any WSGI compliant server, with no additional glue code. Unlike the pre-WSGI versions, this could include WSGI servers that the authors of MoinMoin know nothing about.
The term \textit{MVC} is often encountered in statements such as “framework \textit{foo} supports MVC”. MVC is more about the overall organization of code, rather than any particular API. Many web frameworks use this model to help the developer bring structure to their program. Bigger web applications can have lots of code, so it is a good idea to have an effective structure right from the beginning. That way, even users of other frameworks (or even other languages, since MVC is not Python-specific) can easily understand the code, given that they are already familiar with the MVC structure.

MVC stands for three components:

- The \textit{model}. This is the data that will be displayed and modified. In Python frameworks, this component is often represented by the classes used by an object-relational mapper.
- The \textit{view}. This component’s job is to display the data of the model to the user. Typically this component is implemented via templates.
- The \textit{controller}. This is the layer between the user and the model. The controller reacts to user actions (like opening some specific URL), tells the model to modify the data if necessary, and tells the view code what to display,

While one might think that MVC is a complex design pattern, in fact it is not. It is used in Python because it has turned out to be useful for creating clean, maintainable web sites.

\textbf{Note}: While not all Python frameworks explicitly support MVC, it is often trivial to create a web site which uses the MVC pattern by separating the data logic (the model) from the user interaction logic (the controller) and the templates (the view). That’s why it is
important not to write unnecessary Python code in the templates – it works against the MVC model and creates chaos in the code base, making it harder to understand and modify.

**See also:** The English Wikipedia has an article about the **Model-View-Controller pattern**. It includes a long list of web frameworks for various programming languages.
Ingredients for Websites

Websites are complex constructs, so tools have been created to help web developers make their code easier to write and more maintainable. Tools like these exist for all web frameworks in all languages. Developers are not forced to use these tools, and often there is no “best” tool. It is worth learning about the available tools because they can greatly simplify the process of developing a website.

See also: There are far more components than can be presented here. The Python wiki has a page about these components, called Web Components.

Templates

Mixing of HTML and Python code is made possible by a few libraries. While convenient at first, it leads to horribly unmaintainable code. That's why templates exist. Templates are, in the simplest case, just HTML files with placeholders. The HTML is sent to the user's browser after filling in the placeholders.

Python already includes two ways to build simple templates:

```python
>>> template = "<html><body><h1>Hello %s!</h1></body></html>"
>>> print template % "Reader"
<html><body><h1>Hello Reader!</h1></body></html>
```

```python
>>> from string import Template
>>> template = Template("<html><body><h1>Hello $name</h1></body></html>
```

To generate complex HTML based on non-trivial model data,
conditional and looping constructs like Python’s *for* and *if* are generally needed. *Template engines* support templates of this complexity.

There are a lot of template engines available for Python which can be used with or without a *framework*. Some of these define a plain-text programming language which is easy to learn, partly because it is limited in scope. Others use XML, and the template output is guaranteed to be always be valid XML. There are many other variations.

Some *frameworks* ship their own template engine or recommend one in particular. In the absence of a reason to use a different template engine, using the one provided by or recommended by the framework is a good idea.

Popular template engines include:

- Mako
- Genshi
- Jinja

**See also:** There are many template engines competing for attention, because it is pretty easy to create them in Python. The page *Templating* in the wiki lists a big, ever-growing number of these. The three listed above are considered “second generation” template engines and are a good place to start.

### Data persistence

*Data persistence*, while sounding very complicated, is just about storing data. This data might be the text of blog entries, the postings on a bulletin board or the text of a wiki page. There are, of course, a number of different ways to store information on a web server.
Often, relational database engines like MySQL or PostgreSQL are used because of their good performance when handling very large databases consisting of millions of entries. There is also a small database engine called SQLite, which is bundled with Python in the sqlite3 module, and which uses only one file. It has no other dependencies. For smaller sites SQLite is just enough.

Relational databases are queried using a language called SQL. Python programmers in general do not like SQL too much, as they prefer to work with objects. It is possible to save Python objects into a database using a technology called ORM (Object Relational Mapping). ORM translates all object-oriented access into SQL code under the hood, so the developer does not need to think about it. Most frameworks use ORMs, and it works quite well.

A second possibility is storing data in normal, plain text files (some times called “flat files”). This is very easy for simple sites, but can be difficult to get right if the web site is performing many updates to the stored data.

A third possibility are object oriented databases (also called “object databases”). These databases store the object data in a form that closely parallels the way the objects are structured in memory during program execution. (By contrast, ORMs store the object data as rows of data in tables and relations between those rows.) Storing the objects directly has the advantage that nearly all objects can be saved in a straightforward way, unlike in relational databases where some objects are very hard to represent.

Frameworks often give hints on which data storage method to choose. It is usually a good idea to stick to the data store recommended by the framework unless the application has special requirements better satisfied by an alternate storage mechanism.

See also:
- **Persistence Tools** lists possibilities on how to save data in the file system. Some of these modules are part of the standard library.
- **Database Programming** helps with choosing a method for saving data.
- **SQLAlchemy**, the most powerful OR-Mapper for Python, and **Elixir**, which makes SQLAlchemy easier to use.
- **SQLObject**, another popular OR-Mapper.
- **ZODB** and **Durus**, two object oriented databases.
The process of creating code to run web sites involves writing code to provide various services. The code to provide a particular service often works the same way regardless of the complexity or purpose of the web site in question. Abstracting these common solutions into reusable code produces what are called “frameworks” for web development. Perhaps the most well-known framework for web development is Ruby on Rails, but Python has its own frameworks. Some of these were partly inspired by Rails, or borrowed ideas from Rails, but many existed a long time before Rails.

Originally Python web frameworks tended to incorporate all of the services needed to develop web sites as a giant, integrated set of tools. No two web frameworks were interoperable: a program developed for one could not be deployed on a different one without considerable re-engineering work. This led to the development of “minimalist” web frameworks that provided just the tools to communicate between the Python code and the http protocol, with all other services to be added on top via separate components. Some ad hoc standards were developed that allowed for limited interoperability between frameworks, such as a standard that allowed different template engines to be used interchangeably.

Since the advent of WSGI, the Python web framework world has been evolving toward interoperability based on the WSGI standard. Now many web frameworks, whether “full stack” (providing all the tools one needs to deploy the most complex web sites) or minimalist, or anything in between, are built from collections of reusable components that can be used with more than one framework.

The majority of users will probably want to select a “full stack” framework that has an active community. These frameworks tend to be well documented, and provide the easiest path to producing a
fully functional web site in minimal time.

Some notable frameworks

There are an incredible number of frameworks, so they cannot all be covered here. Instead we will briefly touch on some of the most popular.

Django

Django is a framework consisting of several tightly coupled elements which were written from scratch and work together very well. It includes an ORM which is quite powerful while being simple to use, and has a great online administration interface which makes it possible to edit the data in the database with a browser. The template engine is text-based and is designed to be usable for page designers who cannot write Python. It supports template inheritance and filters (which work like Unix pipes). Django has many handy features bundled, such as creation of RSS feeds or generic views, which make it possible to create web sites almost without writing any Python code.

It has a big, international community, the members of which have created many web sites. There are also a lot of add-on projects which extend Django's normal functionality. This is partly due to Django's well written online documentation and the Django book.

**Note:** Although Django is an MVC-style framework, it names the elements differently, which is described in the Django FAQ.

TurboGears

Another popular web framework for Python is TurboGears. TurboGears takes the approach of using already existing
components and combining them with glue code to create a seamless experience. TurboGears gives the user flexibility in choosing components. For example the ORM and template engine can be changed to use packages different from those used by default.

The documentation can be found in the TurboGears documentation, where links to screencasts can be found. TurboGears has also an active user community which can respond to most related questions. There is also a TurboGears book published, which is a good starting point.

The newest version of TurboGears, version 2.0, moves even further in direction of WSGI support and a component-based architecture. TurboGears 2 is based on the WSGI stack of another popular component-based web framework, Pylons.

### Zope

The Zope framework is one of the “old original” frameworks. Its current incarnation in Zope2 is a tightly integrated full-stack framework. One of its most interesting feature is its tight integration with a powerful object database called the ZODB (Zope Object Database). Because of its highly integrated nature, Zope wound up in a somewhat isolated ecosystem: code written for Zope wasn’t very usable outside of Zope, and vice-versa. To solve this problem the Zope 3 effort was started. Zope 3 re-engineers Zope as a set of more cleanly isolated components. This effort was started before the advent of the WSGI standard, but there is WSGI support for Zope 3 from the Repoze project. Zope components have many years of production use behind them, and the Zope 3 project gives access to these components to the wider Python community. There is even a separate framework based on the Zope components: Grok.

Zope is also the infrastructure used by the Plone content
management system, one of the most powerful and popular content management systems available.

Other notable frameworks

Of course these are not the only frameworks that are available. There are many other frameworks worth mentioning.

Another framework that’s already been mentioned is Pylons. Pylons is much like TurboGears, but with an even stronger emphasis on flexibility, which comes at the cost of being more difficult to use. Nearly every component can be exchanged, which makes it necessary to use the documentation of every single component, of which there are many. Pylons builds upon Paste, an extensive set of tools which are handy for WSGI.

And that’s still not everything. The most up-to-date information can always be found in the Python wiki.

See also: The Python wiki contains an extensive list of web frameworks.

Most frameworks also have their own mailing lists and IRC channels, look out for these on the projects' web sites.
This tutorial is intended to be a gentle introduction to `argparse`, the recommended command-line parsing module in the Python standard library. This was written for argparse in Python 3. A few details are different in 2.x, especially some exception messages, which were improved in 3.x.

**Note:** There are two other modules that fulfill the same task, namely `getopt` (an equivalent for `getopt()` from the C language) and the deprecated `optparse`. Note also that `argparse` is based on `optparse`, and therefore very similar in terms of usage.
Let's show the sort of functionality that we are going to explore in this introductory tutorial by making use of the `ls` command:

```
$ ls
cpython devguide prog.py pypy rm-unused-function.patch
$ ls pypy
ctypes_configure demo dotviewer include lib_pypy lib-python
$ ls -l
```

```
total 20
drwxr-xr-x  19 wena wena   4096 Feb 18 18:51 cpython
drwxr-xr-x   4 wena wena   4096 Feb  8 12:04 devguide
-rwxr-xr-x  1 wena wena    535 Feb 19 00:05 prog.py
drwxr-xr-x  14 wena wena   4096 Feb  7 00:59 pypy
-rw-r--r--  1 wena wena    741 Feb 18 01:01 rm-unused-function.patch
```

```
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by defa
Sort entries alphabetically if none of -cftuvSUX nor --sort is
```

A few concepts we can learn from the four commands:

- The `ls` command is useful when run without any options at all. It defaults to displaying the contents of the current directory.
- If we want beyond what it provides by default, we tell it a bit more. In this case, we want it to display a different directory, `pypy`. What we did is specify what is known as a positional argument. It's named so because the program should know what to do with the value, solely based on where it appears on the command line. This concept is more relevant to a command like `cp`, whose most basic usage is `cp SRC DEST`. The first position is *what you want copied*, and the second position is *where you want it copied to*.
- Now, say we want to change behaviour of the program. In our example, we display more info for each file instead of just
showing the file names. The `-l` in that case is known as an optional argument.

- That's a snippet of the help text. It's very useful in that you can come across a program you have never used before, and can figure out how it works simply by reading its help text.
Let us start with a very simple example which does (almost) nothing:

```python
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

Following is a result of running the code:

```
$ python prog.py
$ python prog.py --help
usage: prog.py [-h]

optional arguments:
  -h, --help    show this help message and exit
$ python prog.py --verbose
usage: prog.py [-h]
prog.py: error: unrecognized arguments: --verbose
$ python prog.py foo
usage: prog.py [-h]
prog.py: error: unrecognized arguments: foo
```

Here is what is happening:

- Running the script without any options results in nothing displayed to stdout. Not so useful.
- The second one starts to display the usefulness of the `argparse` module. We have done almost nothing, but already we get a nice help message.
- The `--help` option, which can also be shortened to `-h`, is the only option we get for free (i.e. no need to specify it). Specifying anything else results in an error. But even then, we do get a useful usage message, also for free.
Introducing Positional arguments

An example:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo")
args = parser.parse_args()
print(args.echo)
```

And running the code:

```
$ python prog.py
usage: prog.py [-h] echo
prog.py: error: the following arguments are required: echo
$ python prog.py --help
usage: prog.py [-h] echo

positional arguments:
  echo

optional arguments:
  -h, --help    show this help message and exit
```

```
$ python prog.py foo
foo
```

Here is what's happening:

- We've added the `add_argument()` method, which is what we use to specify which command-line options the program is willing to accept. In this case, I've named it `echo` so that it's in line with its function.
- Calling our program now requires us to specify an option.
- The `parse_args()` method actually returns some data from the options specified, in this case, `echo`.
- The variable is some form of 'magic' that `argparse` performs for free (i.e. no need to specify which variable that value is stored
You will also notice that its name matches the string argument given to the method, `echo`.

Note however that, although the help display looks nice and all, it currently is not as helpful as it can be. For example we see that we got `echo` as a positional argument, but we don’t know what it does, other than by guessing or by reading the source code. So, let’s make it a bit more useful:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo", help="echo the string you use here")
args = parser.parse_args()
print args.echo
```

And we get:

```
$ python prog.py -h
usage: prog.py [-h] echo

positional arguments:
  echo         echo the string you use here

optional arguments:
  -h, --help    show this help message and exit
```

Now, how about doing something even more useful:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given")
args = parser.parse_args()
print args.square**2
```

Following is a result of running the code:

```
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 5, in <module>
```
That didn’t go so well. That’s because `argparse` treats the options we give it as strings, unless we tell it otherwise. So, let’s tell `argparse` to treat that input as an integer:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square",
                    help="display a square of a given
type=int")
args = parser.parse_args()
print args.square**2
```

Following is a result of running the code:

```
$ python prog.py 4
16
$ python prog.py four
usage: prog.py [-h] square
prog.py: error: argument square: invalid int value: 'four'
```

That went well. The program now even helpfully quits on bad illegal input before proceeding.
Introducing Optional arguments

So far we, have been playing with positional arguments. Let us have a look on how to add optional ones:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity"
args = parser.parse_args()
if args.verbosity:
    print "verbosity turned on"
```

And the output:

```
$ python prog.py --verbosity 1
verbosity turned on
$ python prog.py
$ python prog.py --help
usage: prog.py [-h] [--verbosity VERBOSITY]

optional arguments:
  -h, --help                      show this help message and exit
  --verbosity VERBOSITY           increase output verbosity
$ python prog.py --verbosity
usage: prog.py [-h] [--verbosity VERBOSITY]
prog.py: error: argument --verbosity: expected one argument
```

Here is what is happening:

- The program is written so as to display something when `--verbosity` is specified and display nothing when not.
- To show that the option is actually optional, there is no error when running the program without it. Note that by default, if an optional argument isn’t used, the relevant variable, in this case `args.verbosity`, is given `None` as a value, which is the reason it fails the truth test of the `if` statement.
• The help message is a bit different.
• When using the `--verbosity` option, one must also specify some value, any value.

The above example accepts arbitrary integer values for `--verbosity`, but for our simple program, only two values are actually useful, `True` or `False`. Let’s modify the code accordingly:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbose", help="increase output verbosity", action="store_true")
args = parser.parse_args()
if args.verbose:
    print "verbosity turned on"
```

And the output:

```
$ python prog.py --verbose
verbosity turned on
$ python prog.py --verbose 1
usage: prog.py [-h] [--verbose]
prog.py: error: unrecognized arguments: 1
$ python prog.py --help
usage: prog.py [-h] [--verbose]

optional arguments:
    -h, --help    show this help message and exit
    --verbose    increase output verbosity
```

Here is what is happening:

• The option is now more of a flag than something that requires a value. We even changed the name of the option to match that idea. Note that we now specify a new keyword, `action`, and give it the value `"store_true"`. This means that, if the option is specified, assign the value `True` to `args.verbose`. Not specifying it implies `False`. 
- It complains when you specify a value, in true spirit of what flags actually are.
- Notice the different help text.

**Short options**

If you are familiar with command line usage, you will notice that I haven’t yet touched on the topic of short versions of the options. It’s quite simple:

```python
import argparse
defparser = argparse.ArgumentParser()
defparser.add_argument("-v", "--verbose", help="increase output verbosity", action="store_true")
args = parser.parse_args()
if args.verbose:
    print "verbosity turned on"
```

And here goes:

```
$ python prog.py -v
verbosity turned on
$ python prog.py --help
usage: prog.py [-h] [-v]
optional arguments:
  -h, --help    show this help message and exit
  -v, --verbose increase output verbosity
```

Note that the new ability is also reflected in the help text.
Combining Positional and Optional arguments

Our program keeps growing in complexity:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                     help="display a square of a given number")
parser.add_argument("-v", "--verbose", action="store_true",
                     help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbose:
    print "the square of {} equals {}".format(args.square, answer)
else:
    print answer
```

And now the output:

```
$ python prog.py
usage: prog.py [-h] [-v] square
prog.py: error: the following arguments are required: square
$ python prog.py 4
16
$ python prog.py 4 --verbose
the square of 4 equals 16
$ python prog.py --verbose 4
the square of 4 equals 16
```

- We’ve brought back a positional argument, hence the complaint.
- Note that the order does not matter.

How about we give this program of ours back the ability to have multiple verbosity values, and actually get to use them:

```
import argparse
parser = argparse.ArgumentParser()
```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
        help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int,
        help="increase output verbosity")

args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print "the square of {} equals {}".format(args.square, answer)
elif args.verbosity == 1:
    print "{}^2 == {}".format(args.square, answer)
else:
    print answer

And the output:

$ python prog.py 4
16
$ python prog.py 4 -v
usage: prog.py [-h] [-v VERBOSITY] square
prog.py: error: argument -v/--verbosity: expected one argument
$ python prog.py 4 -v 1
4^2 == 16
$ python prog.py 4 -v 2
the square of 4 equals 16
$ python prog.py 4 -v 3
16

These all look good except the last one, which exposes a bug in our program. Let's fix it by restricting the values the --verbosity option can accept:

import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
        help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int, choices=[0,
        help="increase output verbosity")

args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print "the square of {} equals {}".format(args.square, answer)
elif args.verbosity == 1:
    print "{}^2 == {}".format(args.square, answer)
else:
    print answer

And the output:

$ python prog.py 4 -v 3
usage: prog.py [-h] [-v {0,1,2}] square
prog.py: error: argument -v/--verbosity: invalid choice: 3 (choose from 0, 1, 2)
$ python prog.py 4 -h
usage: prog.py [-h] [-v {0,1,2}] square

positional arguments:
    square                display a square of a given number

optional arguments:
    -h, --help            show this help message and exit
    -v {0,1,2}, --verbosity {0,1,2}
                        increase output verbosity

Note that the change also reflects both in the error message as well as the help string.

Now, let’s use a different approach of playing with verbosity, which is pretty common. It also matches the way the CPython executable handles its own verbosity argument (check the output of python --help):

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display the square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print "the square of {} equals {}".format(args.square, answer)
elif args.verbosity == 1:
    print "{}^2 == {}".format(args.square, answer)
else:
    print answer
```
We have introduced another action, “count”, to count the number of occurrences of a specific optional arguments:

```bash
$ python prog.py 4
16
$ python prog.py 4 -v
4^2 == 16
$ python prog.py 4 -vv
the square of 4 equals 16
$ python prog.py 4 --verbosity --verbosity
the square of 4 equals 16
$ python prog.py 4 -v 1
usage: prog.py [-h] [-v] square
prog.py: error: unrecognized arguments: 1
$ python prog.py 4 -h
usage: prog.py [-h] [-v] square

positional arguments:
  square           display a square of a given number

optional arguments:
  -h, --help       show this help message and exit
  -v, --verbosity  increase output verbosity
$ python prog.py 4 -vvv
16
```

- Yes, it's now more of a flag (similar to `action="store_true"`) in the previous version of our script. That should explain the complaint.
- It also behaves similar to “store_true” action.
- Now here's a demonstration of what the “count” action gives. You've probably seen this sort of usage before.
- And, just like the “store_true” action, if you don't specify the `-v` flag, that flag is considered to have `None` value.
- As should be expected, specifying the long form of the flag, we should get the same output.
- Sadly, our help output isn’t very informative on the new ability our script has acquired, but that can always be fixed by
improving the documentation for our script (e.g. via the `help`
keyword argument).
- That last output exposes a bug in our program.

Let's fix:

```python
import argparse
parser = argparse.ArgumentParser(
    "square", type=int,
    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2

# bugfix: replace == with >=
if args.verbosity >= 2:
    print "the square of {} equals {}".format(args.square, answer)
elif args.verbosity >= 1:
    print "{}^2 == {}".format(args.square, answer)
else:
    print answer
```

And this is what it gives:

```
$ python prog.py 4 -vvv
the square of 4 equals 16
$ python prog.py 4 -vvvv
the square of 4 equals 16
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 11, in <module>
    if args.verbosity >= 2:
TypeError: unorderable types: NoneType() >= int()
```

- First output went well, and fixes the bug we had before. That is,
  we want any value >= 2 to be as verbose as possible.
- Third output not so good.

Let's fix that bug:
```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
        help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count", default
        help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity >= 2:
    print "the square of {} equals {}".format(args.square, answer)
elif args.verbosity >= 1:
    print "{}^2 == {}".format(args.square, answer)
else:
    print answer
```

We've just introduced yet another keyword, `default`. We've set it to `0` in order to make it comparable to the other `int` values. Remember that by default, if an optional argument isn't specified, it gets the `None` value, and that cannot be compared to an `int` value (hence the `TypeError` exception).

And:

```
$ python prog.py 4
16
```

You can go quite far just with what we've learned so far, and we have only scratched the surface. The `argparse` module is very powerful, and we'll explore a bit more of it before we end this tutorial.
Getting a little more advanced

What if we wanted to expand our tiny program to perform other powers, not just squares:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print "{} to the power {} equals {}".format(args.x, args.y, answer)
elif args.verbosity >= 1:
    print "{}^{} == {}".format(args.x, args.y, answer)
else:
    print answer
```

Output:

```
$ python prog.py
usage: prog.py [-h] [x y]
prog.py: error: the following arguments are required: x, y
$ python prog.py -h
usage: prog.py [-h] [x y]

positional arguments:
    x    the base
    y    the exponent

optional arguments:
    -h, --help    show this help message and exit
    -v, --verbosity
$ python prog.py 4 2 -v
4^2 == 16
```

Notice that so far we've been using verbosity level to change the text that gets displayed. The following example instead uses verbosity level to display more text instead:
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print "Running '{}".format(__file__)
if args.verbosity >= 1:
    print "{}^{} ==".format(args.x, args.y),
print answer

Output:

$ python prog.py 4 2
16
$ python prog.py 4 2 -v
4^2 == 16
$ python prog.py 4 2 -vv
Running 'prog.py'
4^2 == 16

Conflicting options

So far, we have been working with two methods of an argparse.ArgumentParser instance. Let's introduce a third one, add_mutually_exclusive_group(). It allows for us to specify options that conflict with each other. Let's also change the rest of the program so that the new functionality makes more sense: we'll introduce the --quiet option, which will be the opposite of the --verbose one:

import argparse
parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
Before we conclude, you probably want to tell your users the main purpose of your program, just in case they don't know:

```
import argparse

import argparse
parser = argparse.ArgumentParser(description="calculate X to the power Y")
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parsed.add_argument("x", type=int, help="the base")
```
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print answer
elif args.verbose:
    print "{} to the power {} equals {}".format(args.x, args.y,
else:
    print "{}^{} == {}".format(args.x, args.y, answer)

Note that slight difference in the usage text. Note the [-v | -q],
which tells us that we can either use -v or -q, but not both at the
same time:

$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x            the base
  y            the exponent

optional arguments:
  -h, --help   show this help message and exit
  -v, --verbose
  -q, --quiet
Conclusion

The `argparse` module offers a lot more than shown here. Its docs are quite detailed and thorough, and full of examples. Having gone through this tutorial, you should easily digest them without feeling overwhelmed.
General Python FAQ

Contents

- General Python FAQ
  - General Information
    - What is Python?
    - What is the Python Software Foundation?
    - Are there copyright restrictions on the use of Python?
    - Why was Python created in the first place?
    - What is Python good for?
    - How does the Python version numbering scheme work?
    - How do I obtain a copy of the Python source?
    - How do I get documentation on Python?
    - I’ve never programmed before. Is there a Python tutorial?
    - Is there a newsgroup or mailing list devoted to Python?
    - How do I get a beta test version of Python?
    - How do I submit bug reports and patches for Python?
    - Are there any published articles about Python that I can reference?
    - Are there any books on Python?
    - Where in the world is www.python.org located?
    - Why is it called Python?
    - Do I have to like “Monty Python’s Flying Circus”?
  - Python in the real world
    - How stable is Python?
    - How many people are using Python?
    - Have any significant projects been done in Python?
    - What new developments are expected for Python in the future?
- Is it reasonable to propose incompatible changes to Python?
- Is Python a good language for beginning programmers?
  - Upgrading Python
    - What is this bsddb185 module my application keeps complaining about?
General Information

What is Python?

Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. It is also usable as an extension language for applications that need a programmable interface. Finally, Python is portable: it runs on many Unix variants, on the Mac, and on PCs under MS-DOS, Windows, Windows NT, and OS/2.

To find out more, start with The Python Tutorial. The Beginner’s Guide to Python links to other introductory tutorials and resources for learning Python.

What is the Python Software Foundation?

The Python Software Foundation is an independent non-profit organization that holds the copyright on Python versions 2.1 and newer. The PSF’s mission is to advance open source technology related to the Python programming language and to publicize the use of Python. The PSF’s home page is at https://www.python.org/psf/.

Donations to the PSF are tax-exempt in the US. If you use Python and find it helpful, please contribute via the PSF donation page.

Are there copyright restrictions on the use of Python?
You can do anything you want with the source, as long as you leave the copyrights in and display those copyrights in any documentation about Python that you produce. If you honor the copyright rules, it’s OK to use Python for commercial use, to sell copies of Python in source or binary form (modified or unmodified), or to sell products that incorporate Python in some form. We would still like to know about all commercial use of Python, of course.

See the PSF license page to find further explanations and a link to the full text of the license.

The Python logo is trademarked, and in certain cases permission is required to use it. Consult the Trademark Usage Policy for more information.

Why was Python created in the first place?

Here’s a very brief summary of what started it all, written by Guido van Rossum:

I had extensive experience with implementing an interpreted language in the ABC group at CWI, and from working with this group I had learned a lot about language design. This is the origin of many Python features, including the use of indentation for statement grouping and the inclusion of very-high-level data types (although the details are all different in Python).

I had a number of gripes about the ABC language, but also liked many of its features. It was impossible to extend the ABC language (or its implementation) to remedy my complaints – in fact its lack of extensibility was one of its biggest problems. I had some experience with using Modula-2+ and talked with the designers of Modula-3 and read the Modula-3 report. Modula-3 is the origin of the syntax and semantics used for exceptions, and some other Python features.
I was working in the Amoeba distributed operating system group at CWI. We needed a better way to do system administration than by writing either C programs or Bourne shell scripts, since Amoeba had its own system call interface which wasn’t easily accessible from the Bourne shell. My experience with error handling in Amoeba made me acutely aware of the importance of exceptions as a programming language feature.

It occurred to me that a scripting language with a syntax like ABC but with access to the Amoeba system calls would fill the need. I realized that it would be foolish to write an Amoeba-specific language, so I decided that I needed a language that was generally extensible.

During the 1989 Christmas holidays, I had a lot of time on my hand, so I decided to give it a try. During the next year, while still mostly working on it in my own time, Python was used in the Amoeba project with increasing success, and the feedback from colleagues made me add many early improvements.

In February 1991, after just over a year of development, I decided to post to USENET. The rest is in the Misc/HISTORY file.

What is Python good for?

Python is a high-level general-purpose programming language that can be applied to many different classes of problems.

The language comes with a large standard library that covers areas such as string processing (regular expressions, Unicode, calculating differences between files), Internet protocols (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, CGI programming), software engineering (unit testing, logging, profiling, parsing Python code), and operating system interfaces (system calls, filesystems, TCP/IP sockets). Look at the table of contents for The Python Standard Library to get an
idea of what's available. A wide variety of third-party extensions are also available. Consult the Python Package Index to find packages of interest to you.

**How does the Python version numbering scheme work?**

Python versions are numbered A.B.C or A.B. A is the major version number – it is only incremented for really major changes in the language. B is the minor version number, incremented for less earth-shattering changes. C is the micro-level – it is incremented for each bugfix release. See PEP 6 for more information about bugfix releases.

Not all releases are bugfix releases. In the run-up to a new major release, a series of development releases are made, denoted as alpha, beta, or release candidate. Alphas are early releases in which interfaces aren't yet finalized; it's not unexpected to see an interface change between two alpha releases. Betas are more stable, preserving existing interfaces but possibly adding new modules, and release candidates are frozen, making no changes except as needed to fix critical bugs.

Alpha, beta and release candidate versions have an additional suffix. The suffix for an alpha version is “aN” for some small number N, the suffix for a beta version is “bN” for some small number N, and the suffix for a release candidate version is “cN” for some small number N. In other words, all versions labeled 2.0aN precede the versions labeled 2.0bN, which precede versions labeled 2.0cN, and *those* precede 2.0.

You may also find version numbers with a “+” suffix, e.g. “2.2+”. These are unreleased versions, built directly from the CPython development repository. In practice, after a final minor release is
made, the version is incremented to the next minor version, which becomes the “a0” version, e.g. “2.4a0”.

See also the documentation for `sys.version`, `sys.hexversion`, and `sys.version_info`.

### How do I obtain a copy of the Python source?

The latest Python source distribution is always available from python.org, at [https://www.python.org/downloads/](https://www.python.org/downloads/). The latest development sources can be obtained via anonymous Mercurial access at [https://hg.python.org/cpython](https://hg.python.org/cpython).

The source distribution is a gzipped tar file containing the complete C source, Sphinx-formatted documentation, Python library modules, example programs, and several useful pieces of freely distributable software. The source will compile and run out of the box on most UNIX platforms.

Consult the [Getting Started section of the Python Developer's Guide](https://docs.python.org/3/) for more information on getting the source code and compiling it.

### How do I get documentation on Python?

The standard documentation for the current stable version of Python is available at [https://docs.python.org/3/](https://docs.python.org/3/). PDF, plain text, and downloadable HTML versions are also available at [https://docs.python.org/3/download.html](https://docs.python.org/3/download.html).

The documentation is written in reStructuredText and processed by the [Sphinx documentation tool](https://sphinx.readthedocs.io). The reStructuredText source for the documentation is part of the Python source distribution.

I’ve never programmed before. Is there a Python
tutorial?

There are numerous tutorials and books available. The standard documentation includes The Python Tutorial.

Consult the Beginner's Guide to find information for beginning Python programmers, including lists of tutorials.

Is there a newsgroup or mailing list devoted to Python?

There is a newsgroup, comp.lang.python, and a mailing list, python-list. The newsgroup and mailing list are gatewayed into each other – if you can read news it's unnecessary to subscribe to the mailing list. comp.lang.python is high-traffic, receiving hundreds of postings every day, and Usenet readers are often more able to cope with this volume.

Announcements of new software releases and events can be found in comp.lang.python.announce, a low-traffic moderated list that receives about five postings per day. It's available as the python-announce mailing list.

More info about other mailing lists and newsgroups can be found at https://www.python.org/community/lists/.

How do I get a beta test version of Python?

Alpha and beta releases are available from https://www.python.org/downloads/. All releases are announced on the comp.lang.python and comp.lang.python.announce newsgroups and on the Python home page at https://www.python.org/; an RSS feed of news is available.
You can also access the development version of Python through Mercurial. See https://docs.python.org/devguide/faq.html for details.

**How do I submit bug reports and patches for Python?**

To report a bug or submit a patch, please use the Roundup installation at https://bugs.python.org/.

You must have a Roundup account to report bugs; this makes it possible for us to contact you if we have follow-up questions. It will also enable Roundup to send you updates as we act on your bug. If you had previously used SourceForge to report bugs to Python, you can obtain your Roundup password through Roundup’s password reset procedure.

For more information on how Python is developed, consult the Python Developer’s Guide.

**Are there any published articles about Python that I can reference?**

It's probably best to cite your favorite book about Python.

The very first article about Python was written in 1991 and is now quite outdated.


**Are there any books on Python?**
Yes, there are many, and more are being published. See the python.org wiki at https://wiki.python.org/moin/PythonBooks for a list.

You can also search online bookstores for “Python” and filter out the Monty Python references; or perhaps search for “Python” and “language”.

**Where in the world is www.python.org located?**

The Python project’s infrastructure is located all over the world. www.python.org is graciously hosted by Rackspace, with CDN caching provided by Fastly. Upfront Systems hosts bugs.python.org. Many other Python services like the Wiki are hosted by Oregon State University Open Source Lab.

**Why is it called Python?**

When he began implementing Python, Guido van Rossum was also reading the published scripts from “Monty Python’s Flying Circus”, a BBC comedy series from the 1970s. Van Rossum thought he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python.

**Do I have to like “Monty Python’s Flying Circus”?**

No, but it helps. :)
Python in the real world

How stable is Python?

Very stable. New, stable releases have been coming out roughly every 6 to 18 months since 1991, and this seems likely to continue. Currently there are usually around 18 months between major releases.

The developers issue “bugfix” releases of older versions, so the stability of existing releases gradually improves. Bugfix releases, indicated by a third component of the version number (e.g. 2.5.3, 2.6.2), are managed for stability; only fixes for known problems are included in a bugfix release, and it’s guaranteed that interfaces will remain the same throughout a series of bugfix releases.

The latest stable releases can always be found on the Python download page. There are two recommended production-ready versions at this point in time, because at the moment there are two branches of stable releases: 2.x and 3.x. Python 3.x may be less useful than 2.x, since currently there is more third party software available for Python 2 than for Python 3. Python 2 code will generally not run unchanged in Python 3.

How many people are using Python?

There are probably tens of thousands of users, though it’s difficult to obtain an exact count.

Python is available for free download, so there are no sales figures, and it’s available from many different sites and packaged with many Linux distributions, so download statistics don’t tell the whole story either.
The comp.lang.python newsgroup is very active, but not all Python users post to the group or even read it.

Have any significant projects been done in Python?

See https://www.python.org/about/success for a list of projects that use Python. Consulting the proceedings for past Python conferences will reveal contributions from many different companies and organizations.

High-profile Python projects include the Mailman mailing list manager and the Zope application server. Several Linux distributions, most notably Red Hat, have written part or all of their installer and system administration software in Python. Companies that use Python internally include Google, Yahoo, and Lucasfilm Ltd.

What new developments are expected for Python in the future?

See https://www.python.org/dev/peps/ for the Python Enhancement Proposals (PEPs). PEPs are design documents describing a suggested new feature for Python, providing a concise technical specification and a rationale. Look for a PEP titled “Python X.Y Release Schedule”, where X.Y is a version that hasn’t been publicly released yet.

New development is discussed on the python-dev mailing list.

Is it reasonable to propose incompatible changes to Python?

In general, no. There are already millions of lines of Python code around the world, so any change in the language that invalidates more than a very small fraction of existing programs has to be
frowned upon. Even if you can provide a conversion program, there’s still the problem of updating all documentation; many books have been written about Python, and we don’t want to invalidate them all at a single stroke.

Providing a gradual upgrade path is necessary if a feature has to be changed. PEP 5 describes the procedure followed for introducing backward-incompatible changes while minimizing disruption for users.

Is Python a good language for beginning programmers?

Yes.

It is still common to start students with a procedural and statically typed language such as Pascal, C, or a subset of C++ or Java. Students may be better served by learning Python as their first language. Python has a very simple and consistent syntax and a large standard library and, most importantly, using Python in a beginning programming course lets students concentrate on important programming skills such as problem decomposition and data type design. With Python, students can be quickly introduced to basic concepts such as loops and procedures. They can probably even work with user-defined objects in their very first course.

For a student who has never programmed before, using a statically typed language seems unnatural. It presents additional complexity that the student must master and slows the pace of the course. The students are trying to learn to think like a computer, decompose problems, design consistent interfaces, and encapsulate data. While learning to use a statically typed language is important in the long term, it is not necessarily the best topic to address in the students’ first programming course.
Many other aspects of Python make it a good first language. Like Java, Python has a large standard library so that students can be assigned programming projects very early in the course that do something. Assignments aren’t restricted to the standard four-function calculator and check balancing programs. By using the standard library, students can gain the satisfaction of working on realistic applications as they learn the fundamentals of programming. Using the standard library also teaches students about code reuse. Third-party modules such as PyGame are also helpful in extending the students’ reach.

Python’s interactive interpreter enables students to test language features while they’re programming. They can keep a window with the interpreter running while they enter their program’s source in another window. If they can’t remember the methods for a list, they can do something like this:

```
>>> L = []
>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr__', '__delvalue__', '__delslice__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> help(L.append)
Help on built-in function append:

append(...)
    L.append(object) -- append object to end

>>> L.append(1)
>>> L
[1]
```

With the interpreter, documentation is never far from the student as
he's programming.

There are also good IDEs for Python. IDLE is a cross-platform IDE for Python that is written in Python using Tkinter. PythonWin is a Windows-specific IDE. Emacs users will be happy to know that there is a very good Python mode for Emacs. All of these programming environments provide syntax highlighting, auto-indenting, and access to the interactive interpreter while coding. Consult the Python wiki for a full list of Python editing environments.

If you want to discuss Python's use in education, you may be interested in joining the edu-sig mailing list.
Upgrading Python

What is this bsddb185 module my application keeps complaining about?

Starting with Python2.3, the distribution includes the PyBSDDB package <http://pybsddb.sf.net/> as a replacement for the old bsddb module. It includes functions which provide backward compatibility at the API level, but requires a newer version of the underlying Berkeley DB library. Files created with the older bsddb module can’t be opened directly using the new module.

Using your old version of Python and a pair of scripts which are part of Python 2.3 (db2pickle.py and pickle2db.py, in the Tools/scripts directory) you can convert your old database files to the new format. Using your old Python version, run the db2pickle.py script to convert it to a pickle, e.g.:

```
python2.2 <pathto>/db2pickle.py database.db database.pck
```

Rename your database file:

```
mv database.db olddatabase.db
```

Now convert the pickle file to a new format database:

```
python <pathto>/pickle2db.py database.db database.pck
```

The precise commands you use will vary depending on the particulars of your installation. For full details about operation of these two scripts check the doc string at the start of each one.
Programming FAQ

Contents

- Programming FAQ
  - General Questions
    - Is there a source code level debugger with breakpoints, single-stepping, etc.?
    - Is there a tool to help find bugs or perform static analysis?
    - How can I create a stand-alone binary from a Python script?
    - Are there coding standards or a style guide for Python programs?
    - My program is too slow. How do I speed it up?
  - Core Language
    - Why am I getting an UnboundLocalError when the variable has a value?
    - What are the rules for local and global variables in Python?
    - Why do lambdas defined in a loop with different values all return the same result?
    - How do I share global variables across modules?
    - What are the “best practices” for using import in a module?
    - Why are default values shared between objects?
    - How can I pass optional or keyword parameters from one function to another?
    - What is the difference between arguments and parameters?
    - Why did changing list ‘y’ also change list ‘x’?
    - How do I write a function with output parameters (call by reference)?
- How do you make a higher order function in Python?
- How do I copy an object in Python?
- How can I find the methods or attributes of an object?
- How can my code discover the name of an object?
- What’s up with the comma operator’s precedence?
- Is there an equivalent of C’s "?:" ternary operator?
- Is it possible to write obfuscated one-liners in Python?
  - Numbers and strings
    - How do I specify hexadecimal and octal integers?
    - Why does -22 // 10 return -3?
    - How do I convert a string to a number?
    - How do I convert a number to a string?
    - How do I modify a string in place?
    - How do I use strings to call functions/methods?
    - Is there an equivalent to Perl's chomp() for removing trailing newlines from strings?
    - Is there a scanf() or sscanf() equivalent?
    - What does ‘UnicodeError: ASCII [decoding,encoding] error: ordinal not in range(128)’ mean?
  - Sequences (Tuples/Lists)
    - How do I convert between tuples and lists?
    - What’s a negative index?
    - How do I iterate over a sequence in reverse order?
    - How do you remove duplicates from a list?
    - How do you make an array in Python?
    - How do I create a multidimensional list?
    - How do I apply a method to a sequence of objects?
    - Why does a_tuple[i] += ['item'] raise an exception when the addition works?
  - Dictionaries
    - How can I get a dictionary to display its keys in a consistent order?
    - I want to do a complicated sort: can you do a
Schwartzian Transform in Python?
- How can I sort one list by values from another list?

Objects
- What is a class?
- What is a method?
- What is self?
- How do I check if an object is an instance of a given class or of a subclass of it?
- What is delegation?
- How do I call a method defined in a base class from a derived class that overrides it?
- How can I organize my code to make it easier to change the base class?
- How do I create static class data and static class methods?
- How can I overload constructors (or methods) in Python?
- I try to use __spam and I get an error about _SomeClassName__spam.
- My class defines __del__ but it is not called when I delete the object.
- How do I get a list of all instances of a given class?
- Why does the result of id() appear to be not unique?

Modules
- How do I create a .pyc file?
- How do I find the current module name?
- How can I have modules that mutually import each other?
- __import__('x.y.z') returns <module 'x'>; how do I get z?
- When I edit an imported module and reimport it, the changes don’t show up. Why does this happen?
Yes.

The `pdb` module is a simple but adequate console-mode debugger for Python. It is part of the standard Python library, and is documented in the Library Reference Manual. You can also write your own debugger by using the code for `pdb` as an example.

The IDLE interactive development environment, which is part of the standard Python distribution (normally available as Tools/scripts/idle), includes a graphical debugger.

PythonWin is a Python IDE that includes a GUI debugger based on `pdb`. The Pythonwin debugger colors breakpoints and has quite a few cool features such as debugging non-Pythonwin programs. Pythonwin is available as part of the Python for Windows Extensions project and as a part of the ActivePython distribution (see https://www.activestate.com/activepython).

**Boa Constructor** is an IDE and GUI builder that uses wxWidgets. It offers visual frame creation and manipulation, an object inspector, many views on the source like object browsers, inheritance hierarchies, doc string generated html documentation, an advanced debugger, integrated help, and Zope support.

**Eric** is an IDE built on PyQt and the Scintilla editing component.

Pydb is a version of the standard Python debugger `pdb`, modified for use with DDD (Data Display Debugger), a popular graphical
There are a number of commercial Python IDEs that include graphical debuggers. They include:

- Wing IDE (https://wingware.com/)
- Komodo IDE (https://komodoide.com/)
- PyCharm (https://www.jetbrains.com/pycharm/)

Is there a tool to help find bugs or perform static analysis?

Yes.

PyChecker is a static analysis tool that finds bugs in Python source code and warns about code complexity and style. You can get PyChecker from http://pychecker.sourceforge.net/.

Pylint is another tool that checks if a module satisfies a coding standard, and also makes it possible to write plug-ins to add a custom feature. In addition to the bug checking that PyChecker performs, Pylint offers some additional features such as checking line length, whether variable names are well-formed according to your coding standard, whether declared interfaces are fully implemented, and more. https://docs.pylint.org/ provides a full list of Pylint's features.

How can I create a stand-alone binary from a Python script?

You don't need the ability to compile Python to C code if all you want is a stand-alone program that users can download and run without
having to install the Python distribution first. There are a number of tools that determine the set of modules required by a program and bind these modules together with a Python binary to produce a single executable.

One is to use the freeze tool, which is included in the Python source tree as `Tools/freeze`. It converts Python byte code to C arrays; a C compiler you can embed all your modules into a new program, which is then linked with the standard Python modules.

It works by scanning your source recursively for import statements (in both forms) and looking for the modules in the standard Python path as well as in the source directory (for built-in modules). It then turns the bytecode for modules written in Python into C code (array initializers that can be turned into code objects using the marshal module) and creates a custom-made config file that only contains those built-in modules which are actually used in the program. It then compiles the generated C code and links it with the rest of the Python interpreter to form a self-contained binary which acts exactly like your script.

Obviously, freeze requires a C compiler. There are several other utilities which don’t. One is Thomas Heller’s py2exe (Windows only) at

http://www.py2exe.org/

Another tool is Anthony Tuininga’s cx_Freeze.

Are there coding standards or a style guide for Python programs?

Yes. The coding style required for standard library modules is documented as PEP 8.
My program is too slow. How do I speed it up?

That’s a tough one, in general. There are many tricks to speed up Python code; consider rewriting parts in C as a last resort.

In some cases it’s possible to automatically translate Python to C or x86 assembly language, meaning that you don’t have to modify your code to gain increased speed.

**Pyrex** can compile a slightly modified version of Python code into a C extension, and can be used on many different platforms.

**Psyco** is a just-in-time compiler that translates Python code into x86 assembly language. If you can use it, Psyco can provide dramatic speedups for critical functions.

The rest of this answer will discuss various tricks for squeezing a bit more speed out of Python code. *Never* apply any optimization tricks unless you know you need them, after profiling has indicated that a particular function is the heavily executed hot spot in the code. Optimizations almost always make the code less clear, and you shouldn’t pay the costs of reduced clarity (increased development time, greater likelihood of bugs) unless the resulting performance benefit is worth it.

There is a page on the wiki devoted to performance tips.

Guido van Rossum has written up an anecdote related to optimization at [https://www.python.org/doc/essays/list2str](https://www.python.org/doc/essays/list2str).

One thing to notice is that function and (especially) method calls are rather expensive; if you have designed a purely OO interface with lots of tiny functions that don’t do much more than get or set an instance variable or call another method, you might consider using a more direct way such as directly accessing instance variables. Also
see the standard module `profile` which makes it possible to find out where your program is spending most of its time (if you have some patience – the profiling itself can slow your program down by an order of magnitude).

Remember that many standard optimization heuristics you may know from other programming experience may well apply to Python. For example it may be faster to send output to output devices using larger writes rather than smaller ones in order to reduce the overhead of kernel system calls. Thus CGI scripts that write all output in “one shot” may be faster than those that write lots of small pieces of output.

Also, be sure to use Python’s core features where appropriate. For example, slicing allows programs to chop up lists and other sequence objects in a single tick of the interpreter’s mainloop using highly optimized C implementations. Thus to get the same effect as:

```python
L2 = []
for i in range(3):
    L2.append(L1[i])
```

it is much shorter and far faster to use

```python
L2 = list(L1[:3])  # "list" is redundant if L1 is a list.
```

Note that the functionally-oriented built-in functions such as `map()`, `zip()`, and friends can be a convenient accelerator for loops that perform a single task. For example to pair the elements of two lists together:

```python
>>> zip([1, 2, 3], [4, 5, 6])
[(1, 4), (2, 5), (3, 6)]
```

or to compute a number of sines:
The operation completes very quickly in such cases.

Other examples include the `join()` and `split()` methods of string objects. For example if `s1..s7` are large (10K+) strings then 

```
''.join([s1, s2, s3, s4, s5, s6, s7])
```

may be far faster than the more obvious `s1+s2+s3+s4+s5+s6+s7`, since the “summation” will compute many subexpressions, whereas `join()` does all the copying in one pass. For manipulating strings, use the `replace()` and the `format()` methods on string objects. Use regular expressions only when you’re not dealing with constant string patterns. You may still use the old `% operations` string % tuple and string % dictionary.

Be sure to use the `list.sort()` built-in method to do sorting, and see the sorting mini-HOWTO for examples of moderately advanced usage. `list.sort()` beats other techniques for sorting in all but the most extreme circumstances.

Another common trick is to “push loops into functions or methods.” For example suppose you have a program that runs slowly and you use the profiler to determine that a Python function `ff()` is being called lots of times. If you notice that `ff()`:

```python
def ff(x):
    ... # do something with x computing result...
    return result
```

tends to be called in loops like:

```python
list = map(ff, oldlist)
```

or:

```python
list = map(lambda x: ff(x), oldlist)
```
for x in sequence:
    value = ff(x)
    ... # do something with value...

then you can often eliminate function call overhead by rewriting `ff()` to:

```python
def ffseq(seq):
    resultseq = []
    for x in seq:
        ... # do something with x computing result...
        resultseq.append(result)
    return resultseq
```

and rewrite the two examples to `list = ffseq(oldlist)` and to:

```python
for value in ffseq(sequence):
    ... # do something with value...
```

Single calls to `ff(x)` translate to `ffseq([x])[0]` with little penalty. Of course this technique is not always appropriate and there are other variants which you can figure out.

You can gain some performance by explicitly storing the results of a function or method lookup into a local variable. A loop like:

```python
for key in token:
    dict[key] = dict.get(key, 0) + 1
```

resolves `dict.get` every iteration. If the method isn't going to change, a slightly faster implementation is:

```python
dict_get = dict.get  # look up the method once
for key in token:
    dict[key] = dict_get(key, 0) + 1
```

Default arguments can be used to determine values once, at compile time instead of at run time. This can only be done for functions or
objects which will not be changed during program execution, such as replacing

```python
def degree_sin(deg):
    return math.sin(deg * math.pi / 180.0)
```

with

```python
def degree_sin(deg, factor=math.pi/180.0, sin=math.sin):
    return sin(deg * factor)
```

Because this trick uses default arguments for terms which should not be changed, it should only be used when you are not concerned with presenting a possibly confusing API to your users.
Core Language

Why am I getting an UnboundLocalError when the variable has a value?

It can be a surprise to get the UnboundLocalError in previously working code when it is modified by adding an assignment statement somewhere in the body of a function.

This code:

```python
>>> x = 10
>>> def bar():
...    print x
>>> bar()
10
```

works, but this code:

```python
>>> x = 10
>>> def foo():
...    print x
...    x += 1
```

results in an UnboundLocalError:

```python
>>> foo()
Traceback (most recent call last):
...  UnboundLocalError: local variable 'x' referenced before assignment
```

This is because when you make an assignment to a variable in a scope, that variable becomes local to that scope and shadows any similarly named variable in the outer scope. Since the last statement in foo assigns a new value to x, the compiler recognizes it as a local
variable. Consequently when the earlier `print x` attempts to print the uninitialized local variable and an error results.

In the example above you can access the outer scope variable by declaring it global:

```python
>>> x = 10
>>> def foobar():
...    global x
...    print x
...    x += 1
>>> foobar()
10
```

This explicit declaration is required in order to remind you that (unlike the superficially analogous situation with class and instance variables) you are actually modifying the value of the variable in the outer scope:

```python
>>> print x
11
```

What are the rules for local and global variables in Python?

In Python, variables that are only referenced inside a function are implicitly global. If a variable is assigned a value anywhere within the function’s body, it’s assumed to be a local unless explicitly declared as global.

Though a bit surprising at first, a moment’s consideration explains this. On one hand, requiring `global` for assigned variables provides a bar against unintended side-effects. On the other hand, if `global` was required for all global references, you’d be using `global` all the time. You’d have to declare as global every reference to a built-in function or to a component of an imported module. This clutter would
defeat the usefulness of the `global` declaration for identifying side-effects.

**Why do lambdas defined in a loop with different values all return the same result?**

Assume you use a for loop to define a few different lambdas (or even plain functions), e.g.:

```python
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
```

This gives you a list that contains 5 lambdas that calculate `x**2`. You might expect that, when called, they would return, respectively, 0, 1, 4, 9, and 16. However, when you actually try you will see that they all return 16:

```python
>>> squares[2]()
16
>>> squares[4]()
16
```

This happens because `x` is not local to the lambdas, but is defined in the outer scope, and it is accessed when the lambda is called — not when it is defined. At the end of the loop, the value of `x` is 4, so all the functions now return `4**2`, i.e. 16. You can also verify this by changing the value of `x` and see how the results of the lambdas change:

```python
>>> x = 8
>>> squares[2]()
64
```

In order to avoid this, you need to save the values in variables local
to the lambdas, so that they don’t rely on the value of the global $x$:

```python
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda n=x: n**2)
```

Here, $n=x$ creates a new variable $n$ local to the lambda and computed when the lambda is defined so that it has the same value that $x$ had at that point in the loop. This means that the value of $n$ will be 0 in the first lambda, 1 in the second, 2 in the third, and so on. Therefore each lambda will now return the correct result:

```python
>>> squares[2]()
4
>>> squares[4]()
16
```

Note that this behaviour is not peculiar to lambdas, but applies to regular functions too.

### How do I share global variables across modules?

The canonical way to share information across modules within a single program is to create a special module (often called config or cfg). Just import the config module in all modules of your application; the module then becomes available as a global name. Because there is only one instance of each module, any changes made to the module object get reflected everywhere. For example:

**config.py:**

```python
x = 0  # Default value of the 'x' configuration setting
```

**mod.py:**

```python
import config
config.x = 1
```
main.py:

```python
import config
import mod
print config.x
```

Note that using a module is also the basis for implementing the Singleton design pattern, for the same reason.

**What are the “best practices” for using import in a module?**

In general, don’t use `from modulename import *`. Doing so clutters the importer’s namespace, and makes it much harder for linters to detect undefined names.

Import modules at the top of a file. Doing so makes it clear what other modules your code requires and avoids questions of whether the module name is in scope. Using one import per line makes it easy to add and delete module imports, but using multiple imports per line uses less screen space.

It’s good practice if you import modules in the following order:

1. standard library modules – e.g. `sys`, `os`, `getopt`, `re`
2. third-party library modules (anything installed in Python’s site-packages directory) – e.g. `mx.DateTime`, `ZODB`, `PIL.Image`, etc.
3. locally-developed modules

Only use explicit relative package imports. If you’re writing code that’s in the `package.sub.m1` module and want to import `package.sub.m2`, do not just write `import m2`, even though it’s legal. Write `from package.sub import m2` or `from . import m2` instead.
It is sometimes necessary to move imports to a function or class to avoid problems with circular imports. Gordon McMillan says:

Circular imports are fine where both modules use the “import <module>” form of import. They fail when the 2nd module wants to grab a name out of the first (“from module import name”) and the import is at the top level. That’s because names in the 1st are not yet available, because the first module is busy importing the 2nd.

In this case, if the second module is only used in one function, then the import can easily be moved into that function. By the time the import is called, the first module will have finished initializing, and the second module can do its import.

It may also be necessary to move imports out of the top level of code if some of the modules are platform-specific. In that case, it may not even be possible to import all of the modules at the top of the file. In this case, importing the correct modules in the corresponding platform-specific code is a good option.

Only move imports into a local scope, such as inside a function definition, if it’s necessary to solve a problem such as avoiding a circular import or are trying to reduce the initialization time of a module. This technique is especially helpful if many of the imports are unnecessary depending on how the program executes. You may also want to move imports into a function if the modules are only ever used in that function. Note that loading a module the first time may be expensive because of the one time initialization of the module, but loading a module multiple times is virtually free, costing only a couple of dictionary lookups. Even if the module name has gone out of scope, the module is probably available in `sys.modules`.

**Why are default values shared between objects?**
This type of bug commonly bites neophyte programmers. Consider this function:

```python
def foo(mydict={}):  # Danger: shared reference to one dict for ...
    # compute something ...
    mydict[key] = value
    return mydict
```

The first time you call this function, `mydict` contains a single item. The second time, `mydict` contains two items because when `foo()` begins executing, `mydict` starts out with an item already in it.

It is often expected that a function call creates new objects for default values. This is not what happens. Default values are created exactly once, when the function is defined. If that object is changed, like the dictionary in this example, subsequent calls to the function will refer to this changed object.

By definition, immutable objects such as numbers, strings, tuples, and `None`, are safe from change. Changes to mutable objects such as dictionaries, lists, and class instances can lead to confusion.

Because of this feature, it is good programming practice to not use mutable objects as default values. Instead, use `None` as the default value and inside the function, check if the parameter is `None` and create a new list/dictionary/whatever if it is. For example, don’t write:

```python
def foo(mydict={}):
    ...
```

but:

```python
def foo(mydict=None):
    if mydict is None:
        mydict = {}  # create a new dict for local namespace
```
This feature can be useful. When you have a function that’s time-consuming to compute, a common technique is to cache the parameters and the resulting value of each call to the function, and return the cached value if the same value is requested again. This is called “memoizing”, and can be implemented like this:

```python
# Callers will never provide a third parameter for this function.
def expensive(arg1, arg2, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]
    # Calculate the value
    result = ... expensive computation ...
    _cache[(arg1, arg2)] = result  # Store result in the cache
    return result
```

You could use a global variable containing a dictionary instead of the default value; it’s a matter of taste.

**How can I pass optional or keyword parameters from one function to another?**

Collect the arguments using the `*` and `**` specifiers in the function’s parameter list; this gives you the positional arguments as a tuple and the keyword arguments as a dictionary. You can then pass these arguments when calling another function by using `*` and `**`:

```python
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

In the unlikely case that you care about Python versions older than 2.0, use `apply()`:

```python
def f(x, *args, **kwargs):
```
What is the difference between arguments and parameters?

Parameters are defined by the names that appear in a function definition, whereas arguments are the values actually passed to a function when calling it. Parameters define what types of arguments a function can accept. For example, given the function definition:

```python
def func(foo, bar=None, **kwargs):
    pass
```

`foo`, `bar` and `kwargs` are parameters of `func`. However, when calling `func`, for example:

```python
func(42, bar=314, extra=somevar)
```

the values 42, 314, and `somevar` are arguments.

Why did changing list ‘y’ also change list ‘x’?

If you wrote code like:

```python
>>> x = []
>>> y = x
>>> y.append(10)
>>> y
[10]
>>> x
[10]
```

you might be wondering why appending an element to `y` changed `x` too.
There are two factors that produce this result:

1. Variables are simply names that refer to objects. Doing \( y = x \) doesn't create a copy of the list – it creates a new variable \( y \) that refers to the same object \( x \) refers to. This means that there is only one object (the list), and both \( x \) and \( y \) refer to it.

2. Lists are mutable, which means that you can change their content.

After the call to `append()`, the content of the mutable object has changed from `[]` to `[10]`. Since both the variables refer to the same object, using either name accesses the modified value `[10]`.

If we instead assign an immutable object to `x`:

```python
>>> x = 5  # ints are immutable
>>> y = x
>>> x = x + 1  # 5 can't be mutated, we are creating a new obj
>>> x
6
>>> y
5
```

we can see that in this case \( x \) and \( y \) are not equal anymore. This is because integers are immutable, and when we do \( x = x + 1 \) we are not mutating the int 5 by incrementing its value; instead, we are creating a new object (the int 6) and assigning it to \( x \) (that is, changing which object \( x \) refers to). After this assignment we have two objects (the ints 6 and 5) and two variables that refer to them (\( x \) now refers to 6 but \( y \) still refers to 5).

Some operations (for example \( y.append(10) \) and \( y.sort() \)) mutate the object, whereas superficially similar operations (for example \( y = y + [10] \) and \( sorted(y) \)) create a new object. In general in Python
(and in all cases in the standard library) a method that mutates an object will return None to help avoid getting the two types of operations confused. So if you mistakenly write y.sort() thinking it will give you a sorted copy of y, you’ll instead end up with None, which will likely cause your program to generate an easily diagnosed error.

However, there is one class of operations where the same operation sometimes has different behaviors with different types: the augmented assignment operators. For example, += mutates lists but not tuples or ints (a_list += [1, 2, 3] is equivalent to a_list.extend([1, 2, 3]) and mutates a_list, whereas some_tuple += (1, 2, 3) and some_int += 1 create new objects).

In other words:

- If we have a mutable object (list, dict, set, etc.), we can use some specific operations to mutate it and all the variables that refer to it will see the change.
- If we have an immutable object (str, int, tuple, etc.), all the variables that refer to it will always see the same value, but operations that transform that value into a new value always return a new object.

If you want to know if two variables refer to the same object or not, you can use the is operator, or the built-in function id().

How do I write a function with output parameters (call by reference)?

Remember that arguments are passed by assignment in Python. Since assignment just creates references to objects, there’s no alias between an argument name in the caller and callee, and so no call-by-reference per se. You can achieve the desired effect in a number
of ways.

1. By returning a tuple of the results:

```python
def func2(a, b):
    a = 'new-value'  # a and b are local names
    b = b + 1        # assigned to new objects
    return a, b      # return new values

x, y = 'old-value', 99
x, y = func2(x, y)
print x, y  # output: new-value 100
```

This is almost always the clearest solution.

2. By using global variables. This isn't thread-safe, and is not recommended.

3. By passing a mutable (changeable in-place) object:

```python
def func1(a):
    a[0] = 'new-value'  # 'a' references a mutable list
    a[1] = a[1] + 1     # changes a shared object

args = ['old-value', 99]
func1(args)
print args[0], args[1]  # output: new-value 100
```

4. By passing in a dictionary that gets mutated:

```python
def func3(args):
    args['a'] = 'new-value'  # args is a mutable diction
    args['b'] = args['b'] + 1  # change it in-place

args = {'a': 'old-value', 'b': 99}
func3(args)
print args['a'], args['b']
```

5. Or bundle up values in a class instance:

```python
class callByRef:
```
```python
def __init__(self, **args):
    for (key, value) in args.items():
        setattr(self, key, value)

def func4(args):
    args.a = 'new-value'  # args is a mutable callByRef
    args.b = args.b + 1   # change object in-place

args = callByRef(a='old-value', b=99)
func4(args)
print args.a, args.b
```

There's almost never a good reason to get this complicated.

Your best choice is to return a tuple containing the multiple results.

How do you make a higher order function in Python?

You have two choices: you can use nested scopes or you can use callable objects. For example, suppose you wanted to define `linear(a, b)` which returns a function $f(x)$ that computes the value $a*x+b$. Using nested scopes:

```python
def linear(a, b):
    def result(x):
        return a * x + b
    return result
```

Or using a callable object:

```python
class linear:
    def __init__(self, a, b):
        self.a, self.b = a, b
    def __call__(self, x):
        return self.a * x + self.b
```
In both cases,

```python
taxes = linear(0.3, 2)
```

gives a callable object where `taxes(10e6) == 0.3 * 10e6 + 2`.

The callable object approach has the disadvantage that it is a bit slower and results in slightly longer code. However, note that a collection of callables can share their signature via inheritance:

```python
class exponential(linear):
    # __init__ inherited
    def __call__(self, x):
        return self.a * (x ** self.b)
```

Object can encapsulate state for several methods:

```python
class counter:
    value = 0

    def set(self, x):
        self.value = x

    def up(self):
        self.value = self.value + 1

    def down(self):
        self.value = self.value - 1

count = counter()
inc, dec, reset = count.up, count.down, count.set
```

Here `inc()`, `dec()` and `reset()` act like functions which share the same counting variable.

**How do I copy an object in Python?**

In general, try `copy.copy()` or `copy.deepcopy()` for the general case.
Not all objects can be copied, but most can.

Some objects can be copied more easily. Dictionaries have a `copy()` method:

```python
newdict = olddict.copy()
```

Sequences can be copied by slicing:

```python
new_l = l[:]
```

---

**How can I find the methods or attributes of an object?**

For an instance `x` of a user-defined class, `dir(x)` returns an alphabetized list of the names containing the instance attributes and methods and attributes defined by its class.

---

**How can my code discover the name of an object?**

Generally speaking, it can’t, because objects don’t really have names. Essentially, assignment always binds a name to a value; The same is true of `def` and `class` statements, but in that case the value is a callable. Consider the following code:

```python
>>> class A:
...   pass
... >>> B = A
>>> a = B()
>>> b = a
>>> print b
<__main__.A instance at 0x16D07CC>
>>> print a
<__main__.A instance at 0x16D07CC>
```

Arguably the class has a name: even though it is bound to two
names and invoked through the name B the created instance is still reported as an instance of class A. However, it is impossible to say whether the instance's name is a or b, since both names are bound to the same value.

Generally speaking it should not be necessary for your code to “know the names” of particular values. Unless you are deliberately writing introspective programs, this is usually an indication that a change of approach might be beneficial.

In comp.lang.python, Fredrik Lundh once gave an excellent analogy in answer to this question:

The same way as you get the name of that cat you found on your porch: the cat (object) itself cannot tell you its name, and it doesn’t really care – so the only way to find out what it's called is to ask all your neighbours (namespaces) if it’s their cat (object)...

....and don’t be surprised if you'll find that it's known by many names, or no name at all!

What’s up with the comma operator’s precedence?

Comma is not an operator in Python. Consider this session:

```python
>>> "a" in "b", "a"
(False, 'a')
```

Since the comma is not an operator, but a separator between expressions the above is evaluated as if you had entered:

```python
("a" in "b"), "a"
```

not:
The same is true of the various assignment operators (=, += etc). They are not truly operators but syntactic delimiters in assignment statements.

Is there an equivalent of C's "?:" ternary operator?

Yes, this feature was added in Python 2.5. The syntax would be as follows:

```
[on_true] if [expression] else [on_false]
x, y = 50, 25
small = x if x < y else y
```

For versions previous to 2.5 the answer would be ‘No’.

Is it possible to write obfuscated one-liners in Python?

Yes. Usually this is done by nesting `lambda` within `lambda`. See the following three examples, due to Ulf Bartelt:

```
# Primes < 1000
print filter(None, map(lambda y:y*reduce(lambda x,y:x*y!=0,
map(lambda x,y=y:y%x, range(2,int(pow(y,0.5)+1))),1),range(2,1000))),

# First 10 Fibonacci numbers
print map(lambda x,f=lambda x,y:(x-1,f)+f(x-2,f)) if x>1 else range(10)

# Mandelbrot set
print (lambda Ru,Ro,Iu,Io,IM,Sx,Sy:reduce(lambda x,y:x+y,map(lambda x,y=x+7,map(lambda x,y=yc,Ru=Ru,Ro=Ro,Sy=Sy,L=lambda yc,Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,Sx=Sx,Sy=Sy:reduce(lambda x,y:x+y,map(lambda x,y=yc,Ru=Ru,Ro=Ro,Fi=F=lambda xc,yc,x,y,k,F=lambda xc,yc,x,y,k,F:(k<=0)or
>=4.0) or 1+f(xc,yc,x*yc-y*yc,2.0*x+yc,k-1,f):f(xc,yc,x,y,k,))
```
Don’t try this at home, kids!
Numbers and strings

How do I specify hexadecimal and octal integers?

To specify an octal digit, precede the octal value with a zero, and then a lower or uppercase “o”. For example, to set the variable “a” to the octal value “10” (8 in decimal), type:

```python
>>> a = 0o10
>>> a
8
```

Hexadecimal is just as easy. Simply precede the hexadecimal number with a zero, and then a lower or uppercase “x”. Hexadecimal digits can be specified in lower or uppercase. For example, in the Python interpreter:

```python
>>> a = 0xa5
>>> a
165
>>> b = 0XB2
>>> b
178
```

Why does -22 // 10 return -3?

It’s primarily driven by the desire that \( \text{i} \% \text{j} \) have the same sign as \( \text{j} \). If you want that, and also want:

\[
\text{i} = (\text{i} // \text{j}) \times \text{j} + (\text{i} \% \text{j})
\]

then integer division has to return the floor. C also requires that identity to hold, and then compilers that truncate \( \text{i} // \text{j} \) need to make \( \text{i} \% \text{j} \) have the same sign as \( \text{i} \).
There are few real use cases for $i \% j$ when $j$ is negative. When $j$ is positive, there are many, and in virtually all of them it's more useful for $i \% j$ to be $\geq 0$. If the clock says 10 now, what did it say 200 hours ago? $-190 \% 12 == 2$ is useful; $-190 \% 12 == -10$ is a bug waiting to bite.

**Note:** On Python 2, $a / b$ returns the same as $a // b$ if `__future__.division` is not in effect. This is also known as “classic” division.

---

**How do I convert a string to a number?**

For integers, use the built-in `int()` type constructor, e.g. `int('144') == 144`. Similarly, `float()` converts to floating-point, e.g. `float('144') == 144.0`.

By default, these interpret the number as decimal, so that `int('0144') == 144` and `int('0x144')` raises `ValueError`. `int(string, base)` takes the base to convert from as a second optional argument, so `int('0x144', 16) == 324`. If the base is specified as 0, the number is interpreted using Python's rules: a leading ‘0’ indicates octal, and ‘0x’ indicates a hex number.

Do not use the built-in function `eval()` if all you need is to convert strings to numbers. `eval()` will be significantly slower and it presents a security risk: someone could pass you a Python expression that might have unwanted side effects. For example, someone could pass `__import__('os').system("rm -rf $HOME")` which would erase your home directory.

`eval()` also has the effect of interpreting numbers as Python expressions, so that e.g. `eval('09')` gives a syntax error because Python regards numbers starting with ‘0’ as octal (base 8).
How do I convert a number to a string?

To convert, e.g., the number 144 to the string ‘144’, use the built-in type constructor `str()`. If you want a hexadecimal or octal representation, use the built-in functions `hex()` or `oct()`. For fancy formatting, see the Format String Syntax section, e.g. `{:04d}`.format(144) yields '0144' and `{:3f}`.format(1/3) yields '0.333'. You may also use the % operator on strings. See the library reference manual for details.

How do I modify a string in place?

You can’t, because strings are immutable. If you need an object with this ability, try converting the string to a list or use the array module:

```python
>>> import io
>>> s = "Hello, world"
>>> a = list(s)
>>> print a
['H', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd']
>>> a[7:] = list("there!")
>>> ''.join(a)
'Hello, there!

>>> import array
>>> a = array.array('c', s)
>>> print a
array('c', 'Hello, world')
>>> a[0] = 'y'; print a
array('c', 'yello, world')
>>> a.tostring()
yello, world
```

How do I use strings to call functions/methods?

There are various techniques.
• The best is to use a dictionary that maps strings to functions. The primary advantage of this technique is that the strings do not need to match the names of the functions. This is also the primary technique used to emulate a case construct:

```python
def a():
    pass
def b():
    pass
dispatch = {'go': a, 'stop': b}  # Note lack of parens for
dispatch[get_input()]()  # Note trailing parens to call fun
```

• Use the built-in function `getattr()`:

```python
import foo
getattr(foo, 'bar')()
```

Note that `getattr()` works on any object, including classes, class instances, modules, and so on.

This is used in several places in the standard library, like this:

```python
class Foo:
    def do_foo(self):
        ...

    def do_bar(self):
        ...

f = getattr(foo_instance, 'do_' + opname)
f()
```

• Use `locals()` or `eval()` to resolve the function name:

```python
def myFunc():
    print "hello"
```
fname = "myFunc"

f = locals()[fname]
f()

f = eval(fname)
f()

Note: Using `eval()` is slow and dangerous. If you don’t have absolute control over the contents of the string, someone could pass a string that resulted in an arbitrary function being executed.

**Is there an equivalent to Perl’s chomp() for removing trailing newlines from strings?**

Starting with Python 2.2, you can use `s.rstrip("\r\n")` to remove all occurrences of any line terminator from the end of the string `s` without removing other trailing whitespace. If the string `s` represents more than one line, with several empty lines at the end, the line terminators for all the blank lines will be removed:

```python
>>> lines = ("line 1 \r\n" ... "\r\n" ... "\r\n")
>>> lines.rstrip("\n\r")
'line 1 '  
```

Since this is typically only desired when reading text one line at a time, using `s.rstrip()` this way works well.

For older versions of Python, there are two partial substitutes:

- If you want to remove all trailing whitespace, use the `rstrip()` method of string objects. This removes all trailing whitespace, not just a single newline.
- Otherwise, if there is only one line in the string `s`, use
Is there a scanf() or sscanf() equivalent?

Not as such.

For simple input parsing, the easiest approach is usually to split the line into whitespace-delimited words using the `split()` method of string objects and then convert decimal strings to numeric values using `int()` or `float()`. `split()` supports an optional “sep” parameter which is useful if the line uses something other than whitespace as a separator.

For more complicated input parsing, regular expressions are more powerful than C’s `sscanf()` and better suited for the task.

What does ‘UnicodeError: ASCII [decoding,encoding] error: ordinal not in range(128)’ mean?

This error indicates that your Python installation can handle only 7-bit ASCII strings. There are a couple ways to fix or work around the problem.

If your programs must handle data in arbitrary character set encodings, the environment the application runs in will generally identify the encoding of the data it is handing you. You need to convert the input to Unicode data using that encoding. For example, a program that handles email or web input will typically find character set encoding information in Content-Type headers. This can then be used to properly convert input data to Unicode. Assuming the string referred to by `value` is encoded as UTF-8:

```python
value = unicode(value, "utf-8")
```
will return a Unicode object. If the data is not correctly encoded as UTF-8, the above call will raise a `UnicodeError` exception.

If you only want strings converted to Unicode which have non-ASCII data, you can try converting them first assuming an ASCII encoding, and then generate Unicode objects if that fails:

```python
try:
    x = unicode(value, "ascii")
except UnicodeError:
    value = unicode(value, "utf-8")
else:
    # value was valid ASCII data
    pass
```

It's possible to set a default encoding in a file called `sitecustomize.py` that's part of the Python library. However, this isn't recommended because changing the Python-wide default encoding may cause third-party extension modules to fail.

Note that on Windows, there is an encoding known as “mbcs”, which uses an encoding specific to your current locale. In many cases, and particularly when working with COM, this may be an appropriate default encoding to use.
Sequences (Tuples/Lists)

How do I convert between tuples and lists?

The type constructor `tuple(seq)` converts any sequence (actually, any iterable) into a tuple with the same items in the same order.

For example, `tuple([1, 2, 3])` yields `(1, 2, 3)` and `tuple('abc')` yields `(a, b, c)`. If the argument is a tuple, it does not make a copy but returns the same object, so it is cheap to call `tuple()` when you aren't sure that an object is already a tuple.

The type constructor `list(seq)` converts any sequence or iterable into a list with the same items in the same order. For example, `list((1, 2, 3))` yields `[1, 2, 3]` and `list('abc')` yields ['a', 'b', 'c']`. If the argument is a list, it makes a copy just like `seq[:]` would.

What’s a negative index?

Python sequences are indexed with positive numbers and negative numbers. For positive numbers 0 is the first index 1 is the second index and so forth. For negative indices -1 is the last index and -2 is the penultimate (next to last) index and so forth. Think of `seq[-n]` as the same as `seq[len(seq)-n]`.

Using negative indices can be very convenient. For example `s[:-1]` is all of the string except for its last character, which is useful for removing the trailing newline from a string.

How do I iterate over a sequence in reverse order?

Use the `reversed()` built-in function, which is new in Python 2.4:
for x in reversed(sequence):
    ... # do something with x ...

This won’t touch your original sequence, but build a new copy with reversed order to iterate over.

With Python 2.3, you can use an extended slice syntax:

for x in sequence[::-1]:
    ... # do something with x ...

How do you remove duplicates from a list?

See the Python Cookbook for a long discussion of many ways to do this:

https://code.activestate.com/recipes/52560/

If you don’t mind reordering the list, sort it and then scan from the end of the list, deleting duplicates as you go:

if mylist:
    mylist.sort()
    last = mylist[-1]
    for i in range(len(mylist)-2, -1, -1):
        if last == mylist[i]:
            del mylist[i]
        else:
            last = mylist[i]

If all elements of the list may be used as dictionary keys (i.e. they are all hashable) this is often faster

    d = {}
    for x in mylist:
        d[x] = 1
    mylist = list(d.keys())

In Python 2.5 and later, the following is possible instead:
mylist = list(set(mylist))

This converts the list into a set, thereby removing duplicates, and then back into a list.

How do you make an array in Python?

Use a list:

"this", 1, "is", "an", "array"

Lists are equivalent to C or Pascal arrays in their time complexity; the primary difference is that a Python list can contain objects of many different types.

The array module also provides methods for creating arrays of fixed types with compact representations, but they are slower to index than lists. Also note that the Numeric extensions and others define array-like structures with various characteristics as well.

To get Lisp-style linked lists, you can emulate cons cells using tuples:

lisp_list = ("like", ("this", ("example", None)) )

If mutability is desired, you could use lists instead of tuples. Here the analogue of lisp car is lisp_list[0] and the analogue of cdr is lisp_list[1]. Only do this if you're sure you really need to, because it's usually a lot slower than using Python lists.

How do I create a multidimensional list?

You probably tried to make a multidimensional array like this:

```python
>>> A = [[None] * 2] * 3
```
This looks correct if you print it:

```python
>>> A
[[None, None], [None, None], [None, None]]
```

But when you assign a value, it shows up in multiple places:

```python
>>> A[0][0] = 5
>>> A
[[5, None], [5, None], [5, None]]
```

The reason is that replicating a list with `*` doesn’t create copies, it only creates references to the existing objects. The `*3` creates a list containing 3 references to the same list of length two. Changes to one row will show in all rows, which is almost certainly not what you want.

The suggested approach is to create a list of the desired length first and then fill in each element with a newly created list:

```python
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
```

This generates a list containing 3 different lists of length two. You can also use a list comprehension:

```python
w, h = 2, 3
A = [[None] * w for i in range(h)]
```

Or, you can use an extension that provides a matrix datatype; `NumPy` is the best known.

**How do I apply a method to a sequence of objects?**

Use a list comprehension:
result = [obj.method() for obj in mylist]

More generically, you can try the following function:

```python
def method_map(objects, method, arguments):
    """method_map([a,b], "meth", (1,2)) gives [a.meth(1,2), b.meth(2,2)]""
    objects = len(objects)
    methods = map(getattr, objects, [method]*nobjects)
    return map(apply, methods, [arguments]*nobjects)
```

Why does a_tuple[i] += ['item'] raise an exception when the addition works?

This is because of a combination of the fact that augmented assignment operators are assignment operators, and the difference between mutable and immutable objects in Python.

This discussion applies in general when augmented assignment operators are applied to elements of a tuple that point to mutable objects, but we'll use a list and += as our exemplar.

If you wrote:

```python
>>> a_tuple = (1, 2)
>>> a_tuple[0] += 1
Traceback (most recent call last):
  ...  
TypeError: 'tuple' object does not support item assignment
```

The reason for the exception should be immediately clear: 1 is added to the object a_tuple[0] points to (1), producing the result object, 2, but when we attempt to assign the result of the computation, 2, to element 0 of the tuple, we get an error because we can’t change what an element of a tuple points to.

Under the covers, what this augmented assignment statement is
doing is approximately this:

```python
>>> result = a_tuple[0] + 1
>>> a_tuple[0] = result
Traceback (most recent call last):
 ... 
TypeError: 'tuple' object does not support item assignment
```

It is the assignment part of the operation that produces the error, since a tuple is immutable.

When you write something like:

```python
>>> a_tuple = (["foo", 'bar'])
>>> a_tuple[0] += ['item']
Traceback (most recent call last):
 ... 
TypeError: 'tuple' object does not support item assignment
```

The exception is a bit more surprising, and even more surprising is the fact that even though there was an error, the append worked:

```python
>>> a_tuple[0]
['foo', 'item']
```

To see why this happens, you need to know that (a) if an object implements an `__iadd__` magic method, it gets called when the `+=` augmented assignment is executed, and its return value is what gets used in the assignment statement; and (b) for lists, `__iadd__` is equivalent to calling `extend` on the list and returning the list. That’s why we say that for lists, `+=` is a “shorthand” for `list.extend`:

```python
>>> a_list = []
>>> a_list += [1]
>>> a_list
[1]
```

This is equivalent to:
>>> result = a_list.__iadd__( [1] )
>>> a_list = result

The object pointed to by a_list has been mutated, and the pointer to the mutated object is assigned back to a_list. The end result of the assignment is a no-op, since it is a pointer to the same object that a_list was previously pointing to, but the assignment still happens.

Thus, in our tuple example what is happening is equivalent to:

```python
>>> result = a_tuple[0].__iadd__( ['item'] )
>>> a_tuple[0] = result
```

```
Traceback (most recent call last):
 ...
TypeError: 'tuple' object does not support item assignment
```

The __iadd__ succeeds, and thus the list is extended, but even though result points to the same object that a_tuple[0] already points to, that final assignment still results in an error, because tuples are immutable.
Dictionaries

How can I get a dictionary to display its keys in a consistent order?

You can’t. Dictionaries store their keys in an unpredictable order, so the display order of a dictionary’s elements will be similarly unpredictable.

This can be frustrating if you want to save a printable version to a file, make some changes and then compare it with some other printed dictionary. In this case, use the `pprint` module to pretty-print the dictionary; the items will be presented in order sorted by the key.

A more complicated solution is to subclass `dict` to create a `SortedDict` class that prints itself in a predictable order. Here’s one simpleminded implementation of such a class:

```python
class SortedDict(dict):
    def __repr__(self):
        keys = sorted(self.keys())
        result = '{{{{}}}}'.format(', '.join(result))
        result = result.format(k, self[k]) for k in keys
        return '{!r}: {!r}'.format(k, self[k])

__str__ = __repr__
```

This will work for many common situations you might encounter, though it’s far from a perfect solution. The largest flaw is that if some values in the dictionary are also dictionaries, their values won’t be presented in any particular order.

I want to do a complicated sort: can you do a Schwartzian Transform in Python?
The technique, attributed to Randal Schwartz of the Perl community, sorts the elements of a list by a metric which maps each element to its “sort value”. In Python, use the `key` argument for the `sort()` function:

```
Isorted = L[:]
Isorted.sort(key=lambda s: int(s[10:15]))
```

How can I sort one list by values from another list?

Merge them into a single list of tuples, sort the resulting list, and then pick out the element you want.

```
>>> list1 = ["what", "I'm", "sorting", "by"]
>>> list2 = ["something", "else", "to", "sort"]
>>> pairs = zip(list1, list2)
>>> pairs
[('what', 'something'), ('I'm', 'else'), ('sorting', 'to'), ('by', 'sort')]
>>> pairs.sort()
>>> result = [ x[1] for x in pairs ]
>>> result
['else', 'sort', 'to', 'something']
```

An alternative for the last step is:

```
>>> result = []
>>> for p in pairs: result.append(p[1])
```

If you find this more legible, you might prefer to use this instead of the final list comprehension. However, it is almost twice as slow for long lists. Why? First, the `append()` operation has to reallocate memory, and while it uses some tricks to avoid doing that each time, it still has to do it occasionally, and that costs quite a bit. Second, the expression “result.append” requires an extra attribute lookup, and third, there’s a speed reduction from having to make all those function calls.
Objects

What is a class?

A class is the particular object type created by executing a class statement. Class objects are used as templates to create instance objects, which embody both the data (attributes) and code (methods) specific to a datatype.

A class can be based on one or more other classes, called its base class(es). It then inherits the attributes and methods of its base classes. This allows an object model to be successively refined by inheritance. You might have a generic Mailbox class that provides basic accessor methods for a mailbox, and subclasses such as MboxMailbox, MaildirMailbox, OutlookMailbox that handle various specific mailbox formats.

What is a method?

A method is a function on some object x that you normally call as x.name(arguments...). Methods are defined as functions inside the class definition:

```python
class C:
    def meth(self, arg):
        return arg * 2 + self.attribute
```

What is self?

Self is merely a conventional name for the first argument of a method. A method defined as meth(self, a, b, c) should be called as x.meth(a, b, c) for some instance x of the class in which the
definition occurs; the called method will think it is called as \texttt{meth(x, a, b, c)}.

See also \textit{Why must ‘self’ be used explicitly in method definitions and calls?}.

How do I check if an object is an instance of a given class or of a subclass of it?

Use the built-in function \texttt{isinstance(obj, cls)}. You can check if an object is an instance of any of a number of classes by providing a tuple instead of a single class, e.g. \texttt{isinstance(obj, (class1, class2, ...))}, and can also check whether an object is one of Python's built-in types, e.g. \texttt{isinstance(obj, str)} or \texttt{isinstance(obj, (int, long, float, complex))}.

Note that most programs do not use \texttt{isinstance()} on user-defined classes very often. If you are developing the classes yourself, a more proper object-oriented style is to define methods on the classes that encapsulate a particular behaviour, instead of checking the object's class and doing a different thing based on what class it is. For example, if you have a function that does something:

```python
def search(obj):
    if isinstance(obj, Mailbox):
        ... # code to search a mailbox
    elif isinstance(obj, Document):
        ... # code to search a document
    elif ...
```

A better approach is to define a \texttt{search()} method on all the classes and just call it:

```python
class Mailbox:
    def search(self):
        ... # code to search a mailbox
```
class Document:
    def search(self):
        ...
        # code to search a document

obj.search()

What is delegation?

Delegation is an object oriented technique (also called a design pattern). Let’s say you have an object \( x \) and want to change the behaviour of just one of its methods. You can create a new class that provides a new implementation of the method you're interested in changing and delegates all other methods to the corresponding method of \( x \).

Python programmers can easily implement delegation. For example, the following class implements a class that behaves like a file but converts all written data to uppercase:

class UpperOut:
    def __init__(self, outfile):
        self._outfile = outfile

    def write(self, s):
        self._outfile.write(s.upper())

    def __getattr__(self, name):
        return getattr(self._outfile, name)

Here the UpperOut class redefines the write() method to convert the argument string to uppercase before calling the underlying self._outfile.write() method. All other methods are delegated to the underlying self._outfile object. The delegation is accomplished via the __getattr__ method; consult the language reference for more information about controlling attribute access.
Note that for more general cases delegation can get trickier. When attributes must be set as well as retrieved, the class must define a __setattr__() method too, and it must do so carefully. The basic implementation of __setattr__() is roughly equivalent to the following:

```python
class X:
    ...
    def __setattr__(self, name, value):
        self.__dict__[name] = value
    ...
```

Most __setattr__() implementations must modify self.__dict__ to store local state for self without causing an infinite recursion.

How do I call a method defined in a base class from a derived class that overrides it?

If you’re using new-style classes, use the built-in super() function:

```python
class Derived(Base):
    def meth(self):
        super(Derived, self).meth()
```

If you’re using classic classes: For a class definition such as class Derived(Base): ... you can call method meth() defined in Base (or one of Base’s base classes) as Base.meth(self, arguments...). Here, Base.meth is an unbound method, so you need to provide the self argument.

How can I organize my code to make it easier to change the base class?

You could define an alias for the base class, assign the real base class to it before your class definition, and use the alias throughout
your class. Then all you have to change is the value assigned to the alias. Incidentally, this trick is also handy if you want to decide dynamically (e.g. depending on availability of resources) which base class to use. Example:

```python
BaseAlias = <real base class>

class Derived(BaseAlias):
    def meth(self):
        BaseAlias.meth(self)
        ...
```

**How do I create static class data and static class methods?**

Both static data and static methods (in the sense of C++ or Java) are supported in Python.

For static data, simply define a class attribute. To assign a new value to the attribute, you have to explicitly use the class name in the assignment:

```python
class C:
    count = 0  # number of times C.__init__ called

    def __init__(self):
        C.count = C.count + 1

    def getcount(self):
        return C.count  # or return self.count

C.count also refers to C.count for any c such that `isinstance(c, C)` holds, unless overridden by c itself or by some class on the base-class search path from c.__class__ back to C.

Caution: within a method of C, an assignment like `self.count = 42` creates a new and unrelated instance named “count” in self's own
dict. Rebinding of a class-static data name must always specify the class whether inside a method or not:

```python
C.count = 314
```

Static methods are possible since Python 2.2:

```python
class C:
    def static(arg1, arg2, arg3):
        # No 'self' parameter!
        ...
    static = staticmethod(static)
```

With Python 2.4's decorators, this can also be written as

```python
class C:
    @staticmethod
    def static(arg1, arg2, arg3):
        # No 'self' parameter!
        ...
```

However, a far more straightforward way to get the effect of a static method is via a simple module-level function:

```python
def getcount():
    return C.count
```

If your code is structured so as to define one class (or tightly related class hierarchy) per module, this supplies the desired encapsulation.

**How can I overload constructors (or methods) in Python?**

This answer actually applies to all methods, but the question usually comes up first in the context of constructors.

In C++ you’d write
```cpp
class C {
    C() { cout << "No arguments\n"; }
    C(int i) { cout << "Argument is " << i << "\n"; }
}
```

In Python you have to write a single constructor that catches all cases using default arguments. For example:

```python
class C:
    def __init__(self, i=None):
        if i is None:
            print "No arguments"
        else:
            print "Argument is", i
```

This is not entirely equivalent, but close enough in practice.

You could also try a variable-length argument list, e.g.

```python
def __init__(self, *args):
    ...
```

The same approach works for all method definitions.

I try to use __spam and I get an error about _SomeClassName__spam.

Variable names with double leading underscores are “mangled” to provide a simple but effective way to define class private variables. Any identifier of the form __spam (at least two leading underscores, at most one trailing underscore) is textually replaced with _classname__spam, where classname is the current class name with any leading underscores stripped.

This doesn’t guarantee privacy: an outside user can still deliberately access the "_classname__spam" attribute, and private values are visible in the object’s __dict__. Many Python programmers never
bother to use private variable names at all.

My class defines \texttt{\_\_del\_\_} but it is not called when I delete the object.

There are several possible reasons for this.

The \texttt{del} statement does not necessarily call \texttt{\_\_del\_\_()} – it simply decrements the object's reference count, and if this reaches zero \texttt{\_\_del\_\_()} is called.

If your data structures contain circular links (e.g. a tree where each child has a parent reference and each parent has a list of children) the reference counts will never go back to zero. Once in a while Python runs an algorithm to detect such cycles, but the garbage collector might run some time after the last reference to your data structure vanishes, so your \texttt{\_\_del\_\_()} method may be called at an inconvenient and random time. This is inconvenient if you're trying to reproduce a problem. Worse, the order in which object's \texttt{\_\_del\_\_()} methods are executed is arbitrary. You can run \texttt{gc.collect()} to force a collection, but there \textit{are} pathological cases where objects will never be collected.

Despite the cycle collector, it's still a good idea to define an explicit \texttt{close()} method on objects to be called whenever you're done with them. The \texttt{close()} method can then remove attributes that refer to subobjects. Don't call \texttt{\_\_del\_\_()} directly – \texttt{\_\_del\_\_()} should call \texttt{close()} and \texttt{close()} should make sure that it can be called more than once for the same object.

Another way to avoid cyclical references is to use the \texttt{weakref} module, which allows you to point to objects without incrementing their reference count. Tree data structures, for instance, should use weak references for their parent and sibling references (if they need
them!)

If the object has ever been a local variable in a function that caught an expression in an except clause, chances are that a reference to the object still exists in that function's stack frame as contained in the stack trace. Normally, calling `sys.exc_clear()` will take care of this by clearing the last recorded exception.

Finally, if your `__del__()` method raises an exception, a warning message is printed to `sys.stderr`.

### How do I get a list of all instances of a given class?

Python does not keep track of all instances of a class (or of a built-in type). You can program the class's constructor to keep track of all instances by keeping a list of weak references to each instance.

### Why does the result of `id()` appear to be not unique?

The `id()` built-in returns an integer that is guaranteed to be unique during the lifetime of the object. Since in CPython, this is the object's memory address, it happens frequently that after an object is deleted from memory, the next freshly created object is allocated at the same position in memory. This is illustrated by this example:

```python
>>> id(1000)
13901272
>>> id(2000)
13901272
```

The two ids belong to different integer objects that are created before, and deleted immediately after execution of the `id()` call. To be sure that objects whose id you want to examine are still alive, create another reference to the object:
>>> a = 1000; b = 2000
>>> id(a)
13901272
>>> id(b)
13891296
How do I create a .pyc file?

When a module is imported for the first time (or when the source is more recent than the current compiled file) a .pyc file containing the compiled code should be created in the same directory as the .py file.

One reason that a .pyc file may not be created is permissions problems with the directory. This can happen, for example, if you develop as one user but run as another, such as if you are testing with a web server. Creation of a .pyc file is automatic if you’re importing a module and Python has the ability (permissions, free space, etc...) to write the compiled module back to the directory.

Running Python on a top level script is not considered an import and no .pyc will be created. For example, if you have a top-level module foo.py that imports another module xyz.py, when you run foo, xyz.pyc will be created since xyz is imported, but no foo.pyc file will be created since foo.py isn’t being imported.

If you need to create foo.pyc – that is, to create a .pyc file for a module that is not imported – you can, using the py_compile and compileall modules.

The py_compile module can manually compile any module. One way is to use the compile() function in that module interactively:

```python
>>> import py_compile
>>> py_compile.compile('foo.py')
```
This will write the `.pyc` to the same location as `foo.py` (or you can override that with the optional parameter `cfile`).

You can also automatically compile all files in a directory or directories using the `compileall` module. You can do it from the shell prompt by running `compileall.py` and providing the path of a directory containing Python files to compile:

```bash
python -m compileall .
```

**How do I find the current module name?**

A module can find out its own module name by looking at the predefined global variable `__name__`. If this has the value `'__main__'`, the program is running as a script. Many modules that are usually used by importing them also provide a command-line interface or a self-test, and only execute this code after checking `__name__`:

```python
def main():
    print 'Running test...'
    ...

if __name__ == '__main__':
    main()
```

**How can I have modules that mutually import each other?**

Suppose you have the following modules:

`foo.py`:

```python
from bar import bar_var
foo_var = 1
```
bar.py:

```python
from foo import foo_var
bar_var = 2
```

The problem is that the interpreter will perform the following steps:

- main imports foo
- Empty globals for foo are created
- foo is compiled and starts executing
- foo imports bar
- Empty globals for bar are created
- bar is compiled and starts executing
- bar imports foo (which is a no-op since there already is a module named foo)
- bar.foo_var = foo.foo_var

The last step fails, because Python isn’t done with interpreting foo yet and the global symbol dictionary for foo is still empty.

The same thing happens when you use `import foo`, and then try to access `foo.foo_var` in global code.

There are (at least) three possible workarounds for this problem.

Guido van Rossum recommends avoiding all uses of `from <module> import ...`, and placing all code inside functions. Initializations of global variables and class variables should use constants or built-in functions only. This means everything from an imported module is referenced as `<module>.<name>`.

Jim Roskind suggests performing steps in the following order in each module:

- exports (globals, functions, and classes that don’t need imported
base classes
- *import* statements
- active code (including globals that are initialized from imported values).

van Rossum doesn’t like this approach much because the imports appear in a strange place, but it does work.

Matthias Urlichs recommends restructuring your code so that the recursive import is not necessary in the first place.

These solutions are not mutually exclusive.

```python
__import__('x.y.z') returns <module 'x'>; how do I get z?
```

Consider using the convenience function `import_module()` from `importlib` instead:

```python
z = importlib.import_module('x.y.z')
```

When I edit an imported module and reimport it, the changes don’t show up. Why does this happen?

For reasons of efficiency as well as consistency, Python only reads the module file on the first time a module is imported. If it didn’t, in a program consisting of many modules where each one imports the same basic module, the basic module would be parsed and re-parsed many times. To force rereading of a changed module, do this:

```python
import modname
reload(modname)
```

Warning: this technique is not 100% fool-proof. In particular, modules
containing statements like

```python
from modname import some_objects
```

will continue to work with the old version of the imported objects. If the module contains class definitions, existing class instances will not be updated to use the new class definition. This can result in the following paradoxical behaviour:

```python
>>> import cls
>>> c = cls.C()  # Create an instance of C
>>> reload(cls)
<module 'cls' from 'cls.pyc'>
>>> isinstance(c, cls.C)  # isinstance is false?!?
False
```

The nature of the problem is made clear if you print out the class objects:

```python
>>> c.__class__
<class cls.C at 0x7352a0>
>>> cls.C
<class cls.C at 0x4198d0>
```
Design and History FAQ
Why does Python use indentation for grouping of statements?

Guido van Rossum believes that using indentation for grouping is extremely elegant and contributes a lot to the clarity of the average Python program. Most people learn to love this feature after a while.

Since there are no begin/end brackets there cannot be a disagreement between grouping perceived by the parser and the human reader. Occasionally C programmers will encounter a fragment of code like this:

```python
if (x <= y)
    x++;  
    y--; 
z++; 
```

Only the `x++` statement is executed if the condition is true, but the indentation leads you to believe otherwise. Even experienced C programmers will sometimes stare at it a long time wondering why `y` is being decremented even for `x > y`.

Because there are no begin/end brackets, Python is much less prone to coding-style conflicts. In C there are many different ways to place the braces. If you’re used to reading and writing code that uses one style, you will feel at least slightly uneasy when reading (or being required to write) another style.

Many coding styles place begin/end brackets on a line by themselves. This makes programs considerably longer and wastes valuable screen space, making it harder to get a good overview of a program. Ideally, a function should fit on one screen (say, 20–30 lines). 20 lines of Python can do a lot more work than 20 lines of C. This is not solely due to the lack of begin/end brackets – the lack of
declarations and the high-level data types are also responsible – but the indentation-based syntax certainly helps.
Why am I getting strange results with simple arithmetic operations?

See the next question.
Why are floating point calculations so inaccurate?

People are often very surprised by results like this:

```python
>>> 1.2 - 1.0
0.19999999999999996
```

and think it is a bug in Python. It’s not. This has nothing to do with Python, but with how the underlying C platform handles floating point numbers, and ultimately with the inaccuracies introduced when writing down numbers as a string of a fixed number of digits.

The internal representation of floating point numbers uses a fixed number of binary digits to represent a decimal number. Some decimal numbers can’t be represented exactly in binary, resulting in small roundoff errors.

In decimal math, there are many numbers that can’t be represented with a fixed number of decimal digits, e.g. $1/3 = 0.3333333333.......$

In base 2, $1/2 = 0.1$, $1/4 = 0.01$, $1/8 = 0.001$, etc. $.2$ equals $2/10$ equals $1/5$, resulting in the binary fractional number $0.001100110011001...$

Floating point numbers only have 32 or 64 bits of precision, so the digits are cut off at some point, and the resulting number is $0.19999999999999996$ in decimal, not $0.2$.

A floating point number’s `repr()` function prints as many digits are necessary to make `eval(repr(f)) == f` true for any float f. The `str()` function prints fewer digits and this often results in the more sensible number that was probably intended:
One of the consequences of this is that it is error-prone to compare the result of some computation to a float with `==`. Tiny inaccuracies may mean that `==` fails. Instead, you have to check that the difference between the two numbers is less than a certain threshold:

```python
epsilon = 0.0000000000001  # Tiny allowed error
expected_result = 0.4

if expected_result - epsilon <= computation() <= expected_result + epsilon:
    ...
```

Please see the chapter on floating point arithmetic in the Python tutorial for more information.
Why are Python strings immutable?

There are several advantages.

One is performance: knowing that a string is immutable means we can allocate space for it at creation time, and the storage requirements are fixed and unchanging. This is also one of the reasons for the distinction between tuples and lists.

Another advantage is that strings in Python are considered as “elemental” as numbers. No amount of activity will change the value 8 to anything else, and in Python, no amount of activity will change the string “eight” to anything else.
Why must ‘self’ be used explicitly in method definitions and calls?

The idea was borrowed from Modula-3. It turns out to be very useful, for a variety of reasons.

First, it’s more obvious that you are using a method or instance attribute instead of a local variable. Reading `self.x` or `self.meth()` makes it absolutely clear that an instance variable or method is used even if you don’t know the class definition by heart. In C++, you can sort of tell by the lack of a local variable declaration (assuming globals are rare or easily recognizable) – but in Python, there are no local variable declarations, so you’d have to look up the class definition to be sure. Some C++ and Java coding standards call for instance attributes to have an `m_` prefix, so this explicitness is still useful in those languages, too.

Second, it means that no special syntax is necessary if you want to explicitly reference or call the method from a particular class. In C++, if you want to use a method from a base class which is overridden in a derived class, you have to use the `::` operator – in Python you can write `baseclass.methodname(self, <argument list>)`. This is particularly useful for `__init__()` methods, and in general in cases where a derived class method wants to extend the base class method of the same name and thus has to call the base class method somehow.

Finally, for instance variables it solves a syntactic problem with assignment: since local variables in Python are (by definition!) those variables to which a value is assigned in a function body (and that aren’t explicitly declared global), there has to be some way to tell the interpreter that an assignment was meant to assign to an instance
variable instead of to a local variable, and it should preferably be syntactic (for efficiency reasons). C++ does this through declarations, but Python doesn’t have declarations and it would be a pity having to introduce them just for this purpose. Using the explicit \texttt{self.var} solves this nicely. Similarly, for using instance variables, having to write \texttt{self.var} means that references to unqualified names inside a method don’t have to search the instance’s directories. To put it another way, local variables and instance variables live in two different namespaces, and you need to tell Python which namespace to use.
Why can’t I use an assignment in an expression?

Many people used to C or Perl complain that they want to use this C idiom:

```c
while (line = readline(f)) {
    // do something with line
}
```

where in Python you’re forced to write this:

```python
while True:
    line = f.readline()
    if not line:
        break
...  # do something with line
```

The reason for not allowing assignment in Python expressions is a common, hard-to-find bug in those other languages, caused by this construct:

```python
if (x = 0) {
    // error handling
}
else {
    // code that only works for nonzero x
}
```

The error is a simple typo: `x = 0`, which assigns 0 to the variable `x`, was written while the comparison `x == 0` is certainly what was intended.

Many alternatives have been proposed. Most are hacks that save some typing but use arbitrary or cryptic syntax or keywords, and fail the simple criterion for language change proposals: it should
intuitively suggest the proper meaning to a human reader who has not yet been introduced to the construct.

An interesting phenomenon is that most experienced Python programmers recognize the while True idiom and don’t seem to be missing the assignment in expression construct much; it’s only newcomers who express a strong desire to add this to the language.

There’s an alternative way of spelling this that seems attractive but is generally less robust than the “while True” solution:

```python
line = f.readline()
while line:
    ...  # do something with line...
    line = f.readline()
```

The problem with this is that if you change your mind about exactly how you get the next line (e.g. you want to change it into sys.stdin.readline()) you have to remember to change two places in your program – the second occurrence is hidden at the bottom of the loop.

The best approach is to use iterators, making it possible to loop through objects using the for statement. For example, in the current version of Python file objects support the iterator protocol, so you can now write simply:

```python
for line in f:
    ...  # do something with line...
```
Why does Python use methods for some functionality (e.g. list.index()) but functions for other (e.g. len(list))?  

The major reason is history. Functions were used for those operations that were generic for a group of types and which were intended to work even for objects that didn’t have methods at all (e.g. tuples). It is also convenient to have a function that can readily be applied to an amorphous collection of objects when you use the functional features of Python (map(), zip() et al).

In fact, implementing len(), max(), min() as a built-in function is actually less code than implementing them as methods for each type. One can quibble about individual cases but it’s a part of Python, and it’s too late to make such fundamental changes now. The functions have to remain to avoid massive code breakage.

**Note:** For string operations, Python has moved from external functions (the string module) to methods. However, len() is still a function.
Why is `join()` a string method instead of a list or tuple method?

Strings became much more like other standard types starting in Python 1.6, when methods were added which give the same functionality that has always been available using the functions of the string module. Most of these new methods have been widely accepted, but the one which appears to make some programmers feel uncomfortable is:

```
"",".join(['1', '2', '4', '8', '16'])
```

which gives the result:

```
"1, 2, 4, 8, 16"
```

There are two common arguments against this usage.

The first runs along the lines of: “It looks really ugly using a method of a string literal (string constant)”, to which the answer is that it might, but a string literal is just a fixed value. If the methods are to be allowed on names bound to strings there is no logical reason to make them unavailable on literals.

The second objection is typically cast as: “I am really telling a sequence to join its members together with a string constant”. Sadly, you aren’t. For some reason there seems to be much less difficulty with having `split()` as a string method, since in that case it is easy to see that

```
"1, 2, 4, 8, 16".split("", "")
```

is an instruction to a string literal to return the substrings delimited by
the given separator (or, by default, arbitrary runs of white space). In this case a Unicode string returns a list of Unicode strings, an ASCII string returns a list of ASCII strings, and everyone is happy.

join() is a string method because in using it you are telling the separator string to iterate over a sequence of strings and insert itself between adjacent elements. This method can be used with any argument which obeys the rules for sequence objects, including any new classes you might define yourself.

Because this is a string method it can work for Unicode strings as well as plain ASCII strings. If join() were a method of the sequence types then the sequence types would have to decide which type of string to return depending on the type of the separator.

If none of these arguments persuade you, then for the moment you can continue to use the join() function from the string module, which allows you to write

```python
string.join([ '1', '2', '4', '8', '16' ], ', ', )
```
How fast are exceptions?

A try/except block is extremely efficient if no exceptions are raised. Actually catching an exception is expensive. In versions of Python prior to 2.0 it was common to use this idiom:

```python
try:
    value = mydict[key]
except KeyError:
    mydict[key] = getvalue(key)
    value = mydict[key]
```

This only made sense when you expected the dict to have the key almost all the time. If that wasn’t the case, you coded it like this:

```python
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

Note: In Python 2.0 and higher, you can code this as `value = mydict.setdefault(key, getvalue(key)).`
Why isn’t there a switch or case statement in Python?

You can do this easily enough with a sequence of `if... elif... else`. There have been some proposals for switch statement syntax, but there is no consensus (yet) on whether and how to do range tests. See PEP 275 for complete details and the current status.

For cases where you need to choose from a very large number of possibilities, you can create a dictionary mapping case values to functions to call. For example:

```python
def function_1(...):
...

functions = {'a': function_1,
              'b': function_2,
              'c': self.method_1,
              ...}

func = functions[value]
func()
```

For calling methods on objects, you can simplify yet further by using the `getattr()` built-in to retrieve methods with a particular name:

```python
def visit_a(self, ...):
...

def dispatch(self, value):
    method_name = 'visit_' + str(value)
    method = getattr(self, method_name)
    method()
```

It’s suggested that you use a prefix for the method names, such as `visit_` in this example. Without such a prefix, if values are coming
from an untrusted source, an attacker would be able to call any method on your object.
Can’t you emulate threads in the interpreter instead of relying on an OS-specific thread implementation?

Answer 1: Unfortunately, the interpreter pushes at least one C stack frame for each Python stack frame. Also, extensions can call back into Python at almost random moments. Therefore, a complete threads implementation requires thread support for C.

Answer 2: Fortunately, there is Stackless Python, which has a completely redesigned interpreter loop that avoids the C stack.
Why can’t lambda expressions contain statements?

Python lambda expressions cannot contain statements because Python’s syntactic framework can’t handle statements nested inside expressions. However, in Python, this is not a serious problem. Unlike lambda forms in other languages, where they add functionality, Python lambdas are only a shorthand notation if you’re too lazy to define a function.

Functions are already first class objects in Python, and can be declared in a local scope. Therefore the only advantage of using a lambda instead of a locally-defined function is that you don’t need to invent a name for the function – but that’s just a local variable to which the function object (which is exactly the same type of object that a lambda expression yields) is assigned!
Can Python be compiled to machine code, C or some other language?

Not easily. Python's high level data types, dynamic typing of objects and run-time invocation of the interpreter (using `eval()` or `exec`) together mean that a “compiled” Python program would probably consist mostly of calls into the Python run-time system, even for seemingly simple operations like `x+1`.

Several projects described in the Python newsgroup or at past Python conferences have shown that this approach is feasible, although the speedups reached so far are only modest (e.g. 2x). Jython uses the same strategy for compiling to Java bytecode. (Jim Hugunin has demonstrated that in combination with whole-program analysis, speedups of 1000x are feasible for small demo programs. See the proceedings from the 1997 Python conference for more information.)

Internally, Python source code is always translated into a bytecode representation, and this bytecode is then executed by the Python virtual machine. In order to avoid the overhead of repeatedly parsing and translating modules that rarely change, this byte code is written into a file whose name ends in "\(\text{.pyc}\)" whenever a module is parsed. When the corresponding .py file is changed, it is parsed and translated again and the .pyc file is rewritten.

There is no performance difference once the .pyc file has been loaded, as the bytecode read from the .pyc file is exactly the same as the bytecode created by direct translation. The only difference is that loading code from a .pyc file is faster than parsing and translating a .py file, so the presence of precompiled .pyc files improves the start-up time of Python scripts. If desired, the Lib/compileall.py module can be used to create valid .pyc files for a
given set of modules.

Note that the main script executed by Python, even if its filename ends in .py, is not compiled to a .pyc file. It is compiled to bytecode, but the bytecode is not saved to a file. Usually main scripts are quite short, so this doesn’t cost much speed.

There are also several programs which make it easier to intermingle Python and C code in various ways to increase performance. See, for example, **Cython**, **Psyco**, **Pyrex**, **PyInline**, **Py2Cmod**, and **Weave**.
How does Python manage memory?

The details of Python memory management depend on the implementation. The standard C implementation of Python uses reference counting to detect inaccessible objects, and another mechanism to collect reference cycles, periodically executing a cycle detection algorithm which looks for inaccessible cycles and deletes the objects involved. The `gc` module provides functions to perform a garbage collection, obtain debugging statistics, and tune the collector's parameters.

Jython relies on the Java runtime so the JVM's garbage collector is used. This difference can cause some subtle porting problems if your Python code depends on the behavior of the reference counting implementation.

Sometimes objects get stuck in tracebacks temporarily and hence are not deallocated when you might expect. Clear the tracebacks with:

```python
import sys
sys.exc_clear()
sys.exc_traceback = sys.last_traceback = None
```

Tracebacks are used for reporting errors, implementing debuggers and related things. They contain a portion of the program state extracted during the handling of an exception (usually the most recent exception).

In the absence of circularities and tracebacks, Python programs do not need to manage memory explicitly.

Why doesn’t Python use a more traditional garbage collection scheme? For one thing, this is not a C standard feature and hence
it's not portable. (Yes, we know about the Boehm GC library. It has bits of assembler code for most common platforms, not for all of them, and although it is mostly transparent, it isn't completely transparent; patches are required to get Python to work with it.)

Traditional GC also becomes a problem when Python is embedded into other applications. While in a standalone Python it's fine to replace the standard malloc() and free() with versions provided by the GC library, an application embedding Python may want to have its own substitute for malloc() and free(), and may not want Python's. Right now, Python works with anything that implements malloc() and free() properly.

In Jython, the following code (which is fine in CPython) will probably run out of file descriptors long before it runs out of memory:

```python
for file in very_long_list_of_files:
    f = open(file)
    c = f.read(1)
```

Using the current reference counting and destructor scheme, each new assignment to f closes the previous file. Using GC, this is not guaranteed. If you want to write code that will work with any Python implementation, you should explicitly close the file or use the `with` statement; this will work regardless of GC:

```python
for file in very_long_list_of_files:
    with open(file) as f:
        c = f.read(1)
```
Why isn’t all memory freed when Python exits?

Objects referenced from the global namespaces of Python modules are not always deallocated when Python exits. This may happen if there are circular references. There are also certain bits of memory that are allocated by the C library that are impossible to free (e.g. a tool like Purify will complain about these). Python is, however, aggressive about cleaning up memory on exit and does try to destroy every single object.

If you want to force Python to delete certain things on deallocation use the `atexit` module to run a function that will force those deletions.
Why are there separate tuple and list data types?

Lists and tuples, while similar in many respects, are generally used in fundamentally different ways. Tuples can be thought of as being similar to Pascal records or C structs; they're small collections of related data which may be of different types which are operated on as a group. For example, a Cartesian coordinate is appropriately represented as a tuple of two or three numbers.

Lists, on the other hand, are more like arrays in other languages. They tend to hold a varying number of objects all of which have the same type and which are operated on one-by-one. For example, `os.listdir('.')` returns a list of strings representing the files in the current directory. Functions which operate on this output would generally not break if you added another file or two to the directory.

Tuples are immutable, meaning that once a tuple has been created, you can't replace any of its elements with a new value. Lists are mutable, meaning that you can always change a list's elements. Only immutable elements can be used as dictionary keys, and hence only tuples and not lists can be used as keys.
How are lists implemented?

Python’s lists are really variable-length arrays, not Lisp-style linked lists. The implementation uses a contiguous array of references to other objects, and keeps a pointer to this array and the array's length in a list head structure.

This makes indexing a list \texttt{a[i]} an operation whose cost is independent of the size of the list or the value of the index.

When items are appended or inserted, the array of references is resized. Some cleverness is applied to improve the performance of appending items repeatedly; when the array must be grown, some extra space is allocated so the next few times don’t require an actual resize.
How are dictionaries implemented?

Python’s dictionaries are implemented as resizable hash tables. Compared to B-trees, this gives better performance for lookup (the most common operation by far) under most circumstances, and the implementation is simpler.

Dictionaries work by computing a hash code for each key stored in the dictionary using the `hash()` built-in function. The hash code varies widely depending on the key; for example, “Python” hashes to -539294296 while “python”, a string that differs by a single bit, hashes to 1142331976. The hash code is then used to calculate a location in an internal array where the value will be stored. Assuming that you’re storing keys that all have different hash values, this means that dictionaries take constant time – O(1), in computer science notation – to retrieve a key. It also means that no sorted order of the keys is maintained, and traversing the array as the `.keys()` and `.items()` do will output the dictionary’s content in some arbitrary jumbled order.
Why must dictionary keys be immutable?

The hash table implementation of dictionaries uses a hash value calculated from the key value to find the key. If the key were a mutable object, its value could change, and thus its hash could also change. But since whoever changes the key object can't tell that it was being used as a dictionary key, it can't move the entry around in the dictionary. Then, when you try to look up the same object in the dictionary it won't be found because its hash value is different. If you tried to look up the old value it wouldn't be found either, because the value of the object found in that hash bin would be different.

If you want a dictionary indexed with a list, simply convert the list to a tuple first; the function \texttt{tuple(L)} creates a tuple with the same entries as the list \texttt{L}. Tuples are immutable and can therefore be used as dictionary keys.

Some unacceptable solutions that have been proposed:

- Hash lists by their address (object ID). This doesn't work because if you construct a new list with the same value it won't be found; e.g.:

  ```python
  mydict = {[1, 2]: '12'}
  print mydict[[1, 2]]
  ```

  would raise a KeyError exception because the id of the \texttt{[1, 2]} used in the second line differs from that in the first line. In other words, dictionary keys should be compared using \texttt{==}, not using \texttt{is}.

- Make a copy when using a list as a key. This doesn't work because the list, being a mutable object, could contain a reference to itself, and then the copying code would run into an
infinite loop.

- Allow lists as keys but tell the user not to modify them. This would allow a class of hard-to-track bugs in programs when you forgot or modified a list by accident. It also invalidates an important invariant of dictionaries: every value in `d.keys()` is usable as a key of the dictionary.

- Mark lists as read-only once they are used as a dictionary key. The problem is that it's not just the top-level object that could change its value; you could use a tuple containing a list as a key. Entering anything as a key into a dictionary would require marking all objects reachable from there as read-only – and again, self-referential objects could cause an infinite loop.

There is a trick to get around this if you need to, but use it at your own risk: You can wrap a mutable structure inside a class instance which has both a `__eq__()` and a `__hash__()` method. You must then make sure that the hash value for all such wrapper objects that reside in a dictionary (or other hash based structure), remain fixed while the object is in the dictionary (or other structure).

class ListWrapper:
    def __init__(self, the_list):
        self.the_list = the_list

    def __eq__(self, other):
        return self.the_list == other.the_list

    def __hash__(self):
        l = self.the_list
        result = 98767 - len(l)*555
        for i, el in enumerate(l):
            try:
                result = result + (hash(el) % 9999999) * 1001 +
            except Exception:
                result = (result % 7777777) + i * 333
        return result
Note that the hash computation is complicated by the possibility that some members of the list may be unhashable and also by the possibility of arithmetic overflow.

Furthermore it must always be the case that if \( o1 == o2 \) (ie \( o1.__eq__(o2) \) is True) then \( \text{hash}(o1) == \text{hash}(o2) \) (ie, \( o1.__hash__() == o2.__hash__() \)), regardless of whether the object is in a dictionary or not. If you fail to meet these restrictions dictionaries and other hash based structures will misbehave.

In the case of ListWrapper, whenever the wrapper object is in a dictionary the wrapped list must not change to avoid anomalies. Don't do this unless you are prepared to think hard about the requirements and the consequences of not meeting them correctly. Consider yourself warned.
Why doesn’t list.sort() return the sorted list?

In situations where performance matters, making a copy of the list just to sort it would be wasteful. Therefore, list.sort() sorts the list in place. In order to remind you of that fact, it does not return the sorted list. This way, you won’t be fooled into accidentally overwriting a list when you need a sorted copy but also need to keep the unsorted version around.

In Python 2.4 a new built-in function – sorted() – has been added. This function creates a new list from a provided iterable, sorts it and returns it. For example, here’s how to iterate over the keys of a dictionary in sorted order:

```python
for key in sorted(mydict):
    ... # do whatever with mydict[key]...
```
How do you specify and enforce an interface spec in Python?

An interface specification for a module as provided by languages such as C++ and Java describes the prototypes for the methods and functions of the module. Many feel that compile-time enforcement of interface specifications helps in the construction of large programs.

Python 2.6 adds an `abc` module that lets you define Abstract Base Classes (ABCs). You can then use `isinstance()` and `issubclass()` to check whether an instance or a class implements a particular ABC. The `collections` module defines a set of useful ABCs such as `Iterable`, `Container`, and `MutableMapping`.

For Python, many of the advantages of interface specifications can be obtained by an appropriate test discipline for components. There is also a tool, PyChecker, which can be used to find problems due to subclassing.

A good test suite for a module can both provide a regression test and serve as a module interface specification and a set of examples. Many Python modules can be run as a script to provide a simple “self test.” Even modules which use complex external interfaces can often be tested in isolation using trivial “stub” emulations of the external interface. The `doctest` and `unittest` modules or third-party test frameworks can be used to construct exhaustive test suites that exercise every line of code in a module.

An appropriate testing discipline can help build large complex applications in Python as well as having interface specifications would. In fact, it can be better because an interface specification cannot test certain properties of a program. For example, the `append()` method is expected to add new elements to the end of
some internal list; an interface specification cannot test that your `append()` implementation will actually do this correctly, but it's trivial to check this property in a test suite.

Writing test suites is very helpful, and you might want to design your code with an eye to making it easily tested. One increasingly popular technique, test-directed development, calls for writing parts of the test suite first, before you write any of the actual code. Of course Python allows you to be sloppy and not write test cases at all.
Why is there no goto?

You can use exceptions to provide a “structured goto” that even works across function calls. Many feel that exceptions can conveniently emulate all reasonable uses of the “go” or “goto” constructs of C, Fortran, and other languages. For example:

```python
class label: pass  # declare a label
try:
    ...
    if condition: raise label()  # goto label
    ...
except label:    # where to goto
    pass
    ...
```

This doesn’t allow you to jump into the middle of a loop, but that’s usually considered an abuse of goto anyway. Use sparingly.
Why can’t raw strings (r-strings) end with a backslash?

More precisely, they can’t end with an odd number of backslashes: the unpaired backslash at the end escapes the closing quote character, leaving an unterminated string.

Raw strings were designed to ease creating input for processors (chiefly regular expression engines) that want to do their own backslash escape processing. Such processors consider an unmatched trailing backslash to be an error anyway, so raw strings disallow that. In return, they allow you to pass on the string quote character by escaping it with a backslash. These rules work well when r-strings are used for their intended purpose.

If you’re trying to build Windows pathnames, note that all Windows system calls accept forward slashes too:

```python
f = open("/mydir/file.txt")  # works fine!
```

If you’re trying to build a pathname for a DOS command, try e.g. one of

```python
dir = r"\this\is\my\dos\dir" "\\"
dir = r"\this\is\my\dos\dir\ "[\:-1]
dir = "\this\is\my\dos\dir\\"
```
Why doesn’t Python have a “with” statement for attribute assignments?

Python has a ‘with’ statement that wraps the execution of a block, calling code on the entrance and exit from the block. Some languages have a construct that looks like this:

```python
with obj:
    a = 1  # equivalent to obj.a = 1
    total = total + 1  # obj.total = obj.total + 1
```

In Python, such a construct would be ambiguous.

Other languages, such as Object Pascal, Delphi, and C++, use static types, so it’s possible to know, in an unambiguous way, what member is being assigned to. This is the main point of static typing – the compiler always knows the scope of every variable at compile time.

Python uses dynamic types. It is impossible to know in advance which attribute will be referenced at runtime. Member attributes may be added or removed from objects on the fly. This makes it impossible to know, from a simple reading, what attribute is being referenced: a local one, a global one, or a member attribute?

For instance, take the following incomplete snippet:

```python
def foo(a):
    with a:
        print x
```

The snippet assumes that “a” must have a member attribute called “x”. However, there is nothing in Python that tells the interpreter this. What should happen if “a” is, let us say, an integer? If there is a global variable named “x”, will it be used inside the with block? As
you see, the dynamic nature of Python makes such choices much harder.

The primary benefit of “with” and similar language features (reduction of code volume) can, however, easily be achieved in Python by assignment. Instead of:

```python
function(args).mydict[index][index].a = 21
function(args).mydict[index][index].b = 42
function(args).mydict[index][index].c = 63
```

write this:

```python
ref = function(args).mydict[index][index]
ref.a = 21
ref.b = 42
ref.c = 63
```

This also has the side-effect of increasing execution speed because name bindings are resolved at run-time in Python, and the second version only needs to perform the resolution once.
Why are colons required for the if/while/def/class statements?

The colon is required primarily to enhance readability (one of the results of the experimental ABC language). Consider this:

```python
if a == b
    print a
```

versus

```python
if a == b:
    print a
```

Notice how the second one is slightly easier to read. Notice further how a colon sets off the example in this FAQ answer; it's a standard usage in English.

Another minor reason is that the colon makes it easier for editors with syntax highlighting; they can look for colons to decide when indentation needs to be increased instead of having to do a more elaborate parsing of the program text.
Why does Python allow commas at the end of lists and tuples?

Python lets you add a trailing comma at the end of lists, tuples, and dictionaries:

```python
[1, 2, 3,]
('a', 'b', 'c',)
d = {
    "A": [1, 5],
    "B": [6, 7],  # last trailing comma is optional but good st.
}
```

There are several reasons to allow this.

When you have a literal value for a list, tuple, or dictionary spread across multiple lines, it’s easier to add more elements because you don’t have to remember to add a comma to the previous line. The lines can also be reordered without creating a syntax error.

Accidentally omitting the comma can lead to errors that are hard to diagnose. For example:

```python
x = ["fee",
     "fie"
   "foo",
   "fum"
]
```

This list looks like it has four elements, but it actually contains three: “fee”, “fiefoo” and “fum”. Always adding the comma avoids this source of error.

Allowing the trailing comma may also make programmatic code generation easier.
Library and Extension FAQ

Contents

- Library and Extension FAQ
  - General Library Questions
    - How do I find a module or application to perform task X?
    - Where is the math.py (socket.py, regex.py, etc.) source file?
    - How do I make a Python script executable on Unix?
    - Is there a curses/termcap package for Python?
    - Is there an equivalent to C’s onexit() in Python?
    - Why don’t my signal handlers work?
  - Common tasks
    - How do I test a Python program or component?
    - How do I create documentation from doc strings?
    - How do I get a single keypress at a time?
  - Threads
    - How do I program using threads?
    - None of my threads seem to run: why?
    - How do I parcel out work among a bunch of worker threads?
    - What kinds of global value mutation are thread-safe?
    - Can’t we get rid of the Global Interpreter Lock?
  - Input and Output
    - How do I delete a file? (And other file questions...)
    - How do I copy a file?
    - How do I read (or write) binary data?
    - I can’t seem to use os.read() on a pipe created with os.popen(); why?
    - How do I run a subprocess with pipes connected to both input and output?
- How do I access the serial (RS232) port?
- Why doesn't closing sys.stdout (stdin, stderr) really close it?
- **Network/Internet Programming**
  - What WWW tools are there for Python?
  - How can I mimic CGI form submission (METHOD=POST)?
  - What module should I use to help with generating HTML?
  - How do I send mail from a Python script?
  - How do I avoid blocking in the connect() method of a socket?
- **Databases**
  - Are there any interfaces to database packages in Python?
  - How do you implement persistent objects in Python?
  - Why is cPickle so slow?
  - If my program crashes with a bsddb (or anydbm) database open, it gets corrupted. How come?
  - I tried to open Berkeley DB file, but bsddb produces bsddb.error: (22, ‘Invalid argument’). Help! How can I restore my data?
- **Mathematics and Numerics**
  - How do I generate random numbers in Python?
General Library Questions

How do I find a module or application to perform task X?

Check the Library Reference to see if there’s a relevant standard library module. (Eventually you’ll learn what’s in the standard library and will be able to skip this step.)

For third-party packages, search the Python Package Index or try Google or another Web search engine. Searching for “Python” plus a keyword or two for your topic of interest will usually find something helpful.

Where is the math.py (socket.py, regex.py, etc.) source file?

If you can’t find a source file for a module it may be a built-in or dynamically loaded module implemented in C, C++ or other compiled language. In this case you may not have the source file or it may be something like `mathmodule.c`, somewhere in a C source directory (not on the Python Path).

There are (at least) three kinds of modules in Python:

1. modules written in Python (.py);

2. modules written in C and dynamically loaded (.dll, .pyd, .so, .sl, etc);

3. modules written in C and linked with the interpreter; to get a list of these, type:
import sys
print sys.builtin_module_names

How do I make a Python script executable on Unix?

You need to do two things: the script file's mode must be executable and the first line must begin with `#!` followed by the path of the Python interpreter.

The first is done by executing `chmod +x scriptfile` or perhaps `chmod 755 scriptfile`.

The second can be done in a number of ways. The most straightforward way is to write

```
#!/usr/local/bin/python
```

as the very first line of your file, using the pathname for where the Python interpreter is installed on your platform.

If you would like the script to be independent of where the Python interpreter lives, you can use the `env` program. Almost all Unix variants support the following, assuming the Python interpreter is in a directory on the user's `PATH`:

```
#!/usr/bin/env python
```

Don't do this for CGI scripts. The `PATH` variable for CGI scripts is often very minimal, so you need to use the actual absolute pathname of the interpreter.

Occasionally, a user's environment is so full that the `usr/bin/env` program fails; or there's no env program at all. In that case, you can try the following hack (due to Alex Rezinsky):
The minor disadvantage is that this defines the script's __doc__ string. However, you can fix that by adding

```python
__doc__ = """...Whatever..."""
```

Is there a curses/termcap package for Python?

For Unix variants the standard Python source distribution comes with a curses module in the Modules subdirectory, though it's not compiled by default. (Note that this is not available in the Windows distribution – there is no curses module for Windows.)

The curses module supports basic curses features as well as many additional functions from ncurses and SYSV curses such as colour, alternative character set support, pads, and mouse support. This means the module isn’t compatible with operating systems that only have BSD curses, but there don’t seem to be any currently maintained OSes that fall into this category.

For Windows: use the consolelib module.

Is there an equivalent to C’s onexit() in Python?

The atexit module provides a register function that is similar to C’s onexit().

Why don't my signal handlers work?

The most common problem is that the signal handler is declared with the wrong argument list. It is called as
handler(signum, frame)

so it should be declared with two arguments:

```python
def handler(signum, frame):
    ...
```
Common tasks

How do I test a Python program or component?

Python comes with two testing frameworks. The `doctest` module finds examples in the docstrings for a module and runs them, comparing the output with the expected output given in the docstring.

The `unittest` module is a fancier testing framework modelled on Java and Smalltalk testing frameworks.

To make testing easier, you should use good modular design in your program. Your program should have almost all functionality encapsulated in either functions or class methods – and this sometimes has the surprising and delightful effect of making the program run faster (because local variable accesses are faster than global accesses). Furthermore the program should avoid depending on mutating global variables, since this makes testing much more difficult to do.

The “global main logic” of your program may be as simple as

```python
if __name__ == "__main__":
    main_logic()
```

at the bottom of the main module of your program.

Once your program is organized as a tractable collection of functions and class behaviours you should write test functions that exercise the behaviours. A test suite that automates a sequence of tests can be associated with each module. This sounds like a lot of work, but since Python is so terse and flexible it's surprisingly easy. You can make coding much more pleasant and fun by writing your test functions in parallel with the “production code”, since this makes it
easy to find bugs and even design flaws earlier.

“Support modules” that are not intended to be the main module of a program may include a self-test of the module.

```python
if __name__ == '__main__':
    self_test()
```

Even programs that interact with complex external interfaces may be tested when the external interfaces are unavailable by using “fake” interfaces implemented in Python.

### How do I create documentation from doc strings?

The `pydoc` module can create HTML from the doc strings in your Python source code. An alternative for creating API documentation purely from docstrings is `epydoc`. `Sphinx` can also include docstring content.

### How do I get a single keypress at a time?

For Unix variants there are several solutions. It’s straightforward to do this using curses, but curses is a fairly large module to learn. Here’s a solution without curses:

```python
import termios, fcntl, sys, os
fd = sys.stdin.fileno()

oldterm = termios.tcgetattr(fd)
newattr = termios.tcgetattr(fd)
termios.tcsetattr(fd, termios.TCSANOW, newattr)

oldflags = fcntl.fcntl(fd, fcntl.F_GETFL)
fcntl.fcntl(fd, fcntl.F_SETFL, oldflags | os.O_NONBLOCK)

try:
    while 1:
```
try:
    c = sys.stdin.read(1)
    print "Got character", repr(c)
except IOError:  pass
finally:
    termios.tcsetattr(fd, termios.TCSAFLUSH, oldterm)
    fcntl.fcntl(fd, fcntl.F_SETFL, oldflags)

You need the `termios` and the `fcntl` module for any of this to work, and I've only tried it on Linux, though it should work elsewhere. In this code, characters are read and printed one at a time.

`termios.tcsetattr()` turns off stdin's echoing and disables canonical mode. `fcntl.fcntl()` is used to obtain stdin's file descriptor flags and modify them for non-blocking mode. Since reading stdin when it is empty results in an `IOError`, this error is caught and ignored.
Threads

How do I program using threads?

Be sure to use the `threading` module and not the `thread` module. The `threading` module builds convenient abstractions on top of the low-level primitives provided by the `thread` module.

Aahz has a set of slides from his threading tutorial that are helpful; see http://www.pythoncraft.com/OSCON2001/.

None of my threads seem to run: why?

As soon as the main thread exits, all threads are killed. Your main thread is running too quickly, giving the threads no time to do any work.

A simple fix is to add a sleep to the end of the program that’s long enough for all the threads to finish:

```python
import threading, time

def thread_task(name, n):
    for i in range(n): print name, i

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10) # <-----------------------------!
```

But now (on many platforms) the threads don’t run in parallel, but appear to run sequentially, one at a time! The reason is that the OS thread scheduler doesn’t start a new thread until the previous thread is blocked.
A simple fix is to add a tiny sleep to the start of the run function:

```python
def thread_task(name, n):
    time.sleep(0.001) # <---------------------!
    for i in range(n): print name, i

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)
```

Instead of trying to guess a good delay value for `time.sleep()`, it's better to use some kind of semaphore mechanism. One idea is to use the `Queue` module to create a queue object, let each thread append a token to the queue when it finishes, and let the main thread read as many tokens from the queue as there are threads.

How do I parcel out work among a bunch of worker threads?

Use the `Queue` module to create a queue containing a list of jobs. The `Queue` class maintains a list of objects and has a `.put(obj)` method that adds items to the queue and a `.get()` method to return them. The class will take care of the locking necessary to ensure that each job is handed out exactly once.

Here's a trivial example:

```python
import threading, Queue, time

# The worker thread gets jobs off the queue. When the queue is empty, it
# assumes there will be no more work and exits.
# (Realistically workers will run until terminated.)
def worker():
    print 'Running worker'
    time.sleep(0.1)
    while True:
        try:
```
arg = q.get(block=False)

except Queue.Empty:
    print 'Worker', threading.currentThread(),
    print 'queue empty'
    break
else:
    print 'Worker', threading.currentThread(),
    print 'running with argument', arg
    time.sleep(0.5)

# Create queue
q = Queue.Queue()

# Start a pool of 5 workers
for i in range(5):
    t = threading.Thread(target=worker, name='worker %i' % (i+1)
    t.start()

# Begin adding work to the queue
for i in range(50):
    q.put(i)

# Give threads time to run
print 'Main thread sleeping'
time.sleep(5)

When run, this will produce the following output:

Running worker
Running worker
Running worker
Running worker
Running worker
Main thread sleeping
Worker <Thread(worker 1, started)> running with argument 0
Worker <Thread(worker 2, started)> running with argument 1
Worker <Thread(worker 3, started)> running with argument 2
Worker <Thread(worker 4, started)> running with argument 3
Worker <Thread(worker 5, started)> running with argument 4
Worker <Thread(worker 1, started)> running with argument 5
...

Consult the module's documentation for more details; the Queue class provides a featureful interface.
What kinds of global value mutation are thread-safe?

A global interpreter lock (GIL) is used internally to ensure that only one thread runs in the Python VM at a time. In general, Python offers to switch among threads only between bytecode instructions; how frequently it switches can be set via `sys.setcheckinterval()`. Each bytecode instruction and therefore all the C implementation code reached from each instruction is therefore atomic from the point of view of a Python program.

In theory, this means an exact accounting requires an exact understanding of the PVM bytecode implementation. In practice, it means that operations on shared variables of built-in data types (ints, lists, dicts, etc) that “look atomic” really are.

For example, the following operations are all atomic (L, L1, L2 are lists, D, D1, D2 are dicts, x, y are objects, i, j are ints):

```python
L.append(x)
L1.extend(L2)
x = L[i]
x = L.pop()
L1[i:j] = L2
L.sort()
x = y
x.field = y
D[x] = y
D1.update(D2)
D.keys()
```

These aren't:

```python
i = i+1
L.append(L[-1])
L[i] = L[j]
D[x] = D[x] + 1
```
Operations that replace other objects may invoke those other objects’ \texttt{\_del\_}() method when their reference count reaches zero, and that can affect things. This is especially true for the mass updates to dictionaries and lists. When in doubt, use a mutex!

Can’t we get rid of the Global Interpreter Lock?

The \texttt{global interpreter lock} (GIL) is often seen as a hindrance to Python's deployment on high-end multiprocessor server machines, because a multi-threaded Python program effectively only uses one CPU, due to the insistence that (almost) all Python code can only run while the GIL is held.

Back in the days of Python 1.5, Greg Stein actually implemented a comprehensive patch set (the “free threading” patches) that removed the GIL and replaced it with fine-grained locking. Unfortunately, even on Windows (where locks are very efficient) this ran ordinary Python code about twice as slow as the interpreter using the GIL. On Linux the performance loss was even worse because pthread locks aren’t as efficient.

Since then, the idea of getting rid of the GIL has occasionally come up but nobody has found a way to deal with the expected slowdown, and users who don’t use threads would not be happy if their code ran at half the speed. Greg's free threading patch set has not been kept up-to-date for later Python versions.

This doesn’t mean that you can’t make good use of Python on multi-CPU machines! You just have to be creative with dividing the work up between multiple \texttt{processes} rather than multiple \texttt{threads}. Judicious use of C extensions will also help; if you use a C extension to perform a time-consuming task, the extension can release the GIL while the thread of execution is in the C code and allow other threads to get some work done.
It has been suggested that the GIL should be a per-interpreter-state lock rather than truly global; interpreters then wouldn’t be able to share objects. Unfortunately, this isn’t likely to happen either. It would be a tremendous amount of work, because many object implementations currently have global state. For example, small integers and short strings are cached; these caches would have to be moved to the interpreter state. Other object types have their own free list; these free lists would have to be moved to the interpreter state. And so on.

And I doubt that it can even be done in finite time, because the same problem exists for 3rd party extensions. It is likely that 3rd party extensions are being written at a faster rate than you can convert them to store all their global state in the interpreter state.

And finally, once you have multiple interpreters not sharing any state, what have you gained over running each interpreter in a separate process?
How do I delete a file? (And other file questions...)

Use `os.remove(filename)` or `os.unlink(filename)`; for documentation, see the `os` module. The two functions are identical; `unlink()` is simply the name of the Unix system call for this function.

To remove a directory, use `os.rmdir()`; use `os.mkdir()` to create one. `os.makedirs(path)` will create any intermediate directories in `path` that don’t exist. `os.removedirs(path)` will remove intermediate directories as long as they’re empty; if you want to delete an entire directory tree and its contents, use `shutil.rmtree()`.

To rename a file, use `os.rename(old_path, new_path)`.

To truncate a file, open it using `f = open(filename, "r+"), and use `f.truncate(offset)`; offset defaults to the current seek position. There’s also `os.ftruncate(fd, offset)` for files opened with `os.open()`, where `fd` is the file descriptor (a small integer).

The `shutil` module also contains a number of functions to work on files including `copyfile()`, `copytree()`, and `rmtree()`.

How do I copy a file?

The `shutil` module contains a `copyfile()` function. Note that on MacOS 9 it doesn’t copy the resource fork and Finder info.

How do I read (or write) binary data?

To read or write complex binary data formats, it’s best to use the
**struct** module. It allows you to take a string containing binary data (usually numbers) and convert it to Python objects; and vice versa.

For example, the following code reads two 2-byte integers and one 4-byte integer in big-endian format from a file:

```python
import struct
f = open(filename, "rb")  # Open in binary mode for portability
s = f.read(8)
x, y, z = struct.unpack(">hhh", s)
```

The ‘>’ in the format string forces big-endian data; the letter ‘h’ reads one “short integer” (2 bytes), and ‘l’ reads one “long integer” (4 bytes) from the string.

For data that is more regular (e.g. a homogeneous list of ints or floats), you can also use the **array** module.

I can’t seem to use `os.read()` on a pipe created with `os.popen()`; why?

`os.read()` is a low-level function which takes a file descriptor, a small integer representing the opened file. `os.popen()` creates a high-level file object, the same type returned by the built-in `open()` function. Thus, to read *n* bytes from a pipe *p* created with `os.popen()`, you need to use `p.read(n)`.

How do I run a subprocess with pipes connected to both input and output?

Use the **popen2** module. For example:

```python
import popen2
```
Warning: in general it is unwise to do this because you can easily cause a deadlock where your process is blocked waiting for output from the child while the child is blocked waiting for input from you. This can be caused by the parent expecting the child to output more text than it does or by data being stuck in stdio buffers due to lack of flushing. The Python parent can of course explicitly flush the data it sends to the child before it reads any output, but if the child is a naive C program it may have been written to never explicitly flush its output, even if it is interactive, since flushing is normally automatic.

Note that a deadlock is also possible if you use `popen3()` to read stdout and stderr. If one of the two is too large for the internal buffer (increasing the buffer size does not help) and you `read()` the other one first, there is a deadlock, too.

Note on a bug in popen2: unless your program calls `wait()` or `waitpid()`, finished child processes are never removed, and eventually calls to popen2 will fail because of a limit on the number of child processes. Calling `os.waitpid()` with the `os.WNOHANG` option can prevent this; a good place to insert such a call would be before calling `popen2` again.

In many cases, all you really need is to run some data through a command and get the result back. Unless the amount of data is very large, the easiest way to do this is to write it to a temporary file and run the command with that temporary file as input. The standard module `tempfile` exports a `mktemp()` function to generate unique temporary file names.
class Popen3:
    ""
    This is a deadlock-safe version of popen that returns an object with errorlevel, out (a string) and err (a string (capturestderr may not work under windows.)
    Example: print Popen3('grep spam','\n\nhere spam\n\n').out
    ""
    def __init__(self, command, input=None, capturestderr=None):
        outfile = tempfile.mktemp()
        command = "( %s ) > %s" % (command, outfile)
        if input:
            infile = tempfile.mktemp()
            open(infile, "w").write(input)
            command = command + " <" + infile
        if capturestderr:
            errfile = tempfile.mktemp()
            command = command + " 2>" + errfile
            self.errorlevel = os.system(command) >> 8
        self.out = open(outfile, "r").read()
        os.remove(outfile)
        if input:
            os.remove(infile)
        if capturestderr:
            self.err = open(errfile, "r").read()
            os.remove(errfile)

Note that many interactive programs (e.g. vi) don't work well with pipes substituted for standard input and output. You will have to use pseudo ttys (“ptys”) instead of pipes. Or you can use a Python interface to Don Libes' “expect” library. A Python extension that interfaces to expect is called “expy” and available from http://expectpy.sourceforge.net. A pure Python solution that works like expect is pexpect.

How do I access the serial (RS232) port?

For Win32, POSIX (Linux, BSD, etc.), Jython:

http://pyserial.sourceforge.net
For Unix, see a Usenet post by Mitch Chapman:

https://groups.google.com/groups?selm=34A04430.CF9@ohioee.com

**Why doesn’t closing sys.stdout (stdin, stderr) really close it?**

Python file objects are a high-level layer of abstraction on top of C streams, which in turn are a medium-level layer of abstraction on top of (among other things) low-level C file descriptors.

For most file objects you create in Python via the built-in `file` constructor, `f.close()` marks the Python file object as being closed from Python’s point of view, and also arranges to close the underlying C stream. This also happens automatically in `f`'s destructor, when `f` becomes garbage.

But stdin, stdout and stderr are treated specially by Python, because of the special status also given to them by C. Running `sys.stdout.close()` marks the Python-level file object as being closed, but does *not* close the associated C stream.

To close the underlying C stream for one of these three, you should first be sure that’s what you really want to do (e.g., you may confuse extension modules trying to do I/O). If it is, use `os.close`:

```
  os.close(0)  # close C's stdin stream
  os.close(1)  # close C's stdout stream
  os.close(2)  # close C's stderr stream
```
Network/Internet Programming

What WWW tools are there for Python?

See the chapters titled Internet Protocols and Support and Internet Data Handling in the Library Reference Manual. Python has many modules that will help you build server-side and client-side web systems.

A summary of available frameworks is maintained by Paul Boddie at https://wiki.python.org/moin/WebProgramming.


How can I mimic CGI form submission (METHOD=POST)?

I would like to retrieve web pages that are the result of POSTing a form. Is there existing code that would let me do this easily?

Yes. Here's a simple example that uses httplib:

```python
#!/usr/local/bin/python

import httpplib, sys, time

# build the query string
qs = "First=Josephine&MI=Q&Last=Public"

# connect and send the server a path
httpobj = httpplib.HTTP('www.some-server.out-there', 80)
httpobj.putrequest('POST', '/cgi-bin/some-cgi-script')
# now generate the rest of the HTTP headers...
httpobj.putheader('Accept', '/*/*')
```
httpobj.putheader('Connection', 'Keep-Alive')
httpobj.putheader('Content-type', 'application/x-www-form-urlencoded')
httpobj.putheader('Content-length', '%d' % len(qs))
httpobj.endheaders()
httpobj.send(qs)

# find out what the server said in response...
reply, msg, hdrs = httpobj.getreply()
if reply != 200:
    sys.stdout.write(httpobj.getfile().read())

Note that in general for percent-encoded POST operations, query strings must be quoted using urllib.urlencode(). For example, to send name=Guy Steele, Jr.:

```python
>>> import urllib
>>> urllib.urlencode({'name': 'Guy Steele, Jr.'})
'name=Guy+Steele%2C+Jr.'
```

What module should I use to help with generating HTML?

You can find a collection of useful links on the Web Programming wiki page.

How do I send mail from a Python script?

Use the standard library module smtplib.

Here's a very simple interactive mail sender that uses it. This method will work on any host that supports an SMTP listener.

```python
import sys, smtplib
fromaddr = raw_input("From: ")
toaddrs = raw_input("To: ").split(',,')
print "Enter message, end with ^D:" msg = '
while True:
```
```python
line = sys.stdin.readline()
if not line:
    break
msg += line

# The actual mail send
server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

A Unix-only alternative uses sendmail. The location of the sendmail program varies between systems; sometimes it is `/usr/lib/sendmail`, sometimes `/usr/sbin/sendmail`. The sendmail manual page will help you out. Here's some sample code:

```python
import os
SENDMAIL = "/usr/sbin/sendmail"  # sendmail location
p = os.popen("%s -t -i" % SENDMAIL, "w")
p.write("To: receiver@example.com\n")
p.write("Subject: test\n")
p.write("\n"  # blank line separating headers from body
p.write("Some text\n")
p.write("some more text\n")
sts = p.close()
if sts != 0:
    print "Sendmail exit status", sts
```

How do I avoid blocking in the connect() method of a socket?

The select module is commonly used to help with asynchronous I/O on sockets.

To prevent the TCP connect from blocking, you can set the socket to non-blocking mode. Then when you do the `connect()`, you will either connect immediately (unlikely) or get an exception that contains the error number as `.errno`. `.errno.EINPROGRESS` indicates that the connection is in progress, but hasn’t finished yet. Different OSes will
return different values, so you’re going to have to check what’s returned on your system.

You can use the `connect_ex()` method to avoid creating an exception. It will just return the `errno` value. To poll, you can call `connect_ex()` again later – 0 or `errno.EISCONN` indicate that you’re connected – or you can pass this socket to `select` to check if it’s writable.
Databases

Are there any interfaces to database packages in Python?

Yes.

Python 2.3 includes the **bsddb** package which provides an interface to the BerkeleyDB library. Interfaces to disk-based hashes such as **DBM** and **GDBM** are also included with standard Python.

Support for most relational databases is available. See the [DatabaseProgramming wiki page](https://wiki.python.org/moin/DatabaseProgramming) for details.

How do you implement persistent objects in Python?

The **pickle** library module solves this in a very general way (though you still can't store things like open files, sockets or windows), and the **shelve** library module uses pickle and (g)dbm to create persistent mappings containing arbitrary Python objects. For better performance, you can use the **cPickle** module.

A more awkward way of doing things is to use pickle's little sister, marshal. The **marshal** module provides very fast ways to store noncircular basic Python types to files and strings, and back again. Although marshal does not do fancy things like store instances or handle shared references properly, it does run extremely fast. For example, loading a half megabyte of data may take less than a third of a second. This often beats doing something more complex and general such as using gdbm with pickle/shelve.
Why is cPickle so slow?

By default `pickle` uses a relatively old and slow format for backward compatibility. You can however specify other protocol versions that are faster:

```python
largeString = 'z' * (100 * 1024)
myPickle = cPickle.dumps(largeString, protocol=1)
```

If my program crashes with a bsddb (or anydbm) database open, it gets corrupted. How come?

Databases opened for write access with the bsddb module (and often by the anydbm module, since it will preferentially use bsddb) must explicitly be closed using the `.close()` method of the database. The underlying library caches database contents which need to be converted to on-disk form and written.

If you have initialized a new bsddb database but not written anything to it before the program crashes, you will often wind up with a zero-length file and encounter an exception the next time the file is opened.

I tried to open Berkeley DB file, but bsddb produces bsddb.error: (22, ‘Invalid argument’). Help! How can I restore my data?

Don’t panic! Your data is probably intact. The most frequent cause for the error is that you tried to open an earlier Berkeley DB file with a later version of the Berkeley DB library.

Many Linux systems now have all three versions of Berkeley DB available. If you are migrating from version 1 to a newer version use
db_dump185 to dump a plain text version of the database. If you are migrating from version 2 to version 3 use db2_dump to create a plain text version of the database. In either case, use db_load to create a new native database for the latest version installed on your computer. If you have version 3 of Berkeley DB installed, you should be able to use db2_load to create a native version 2 database.

You should move away from Berkeley DB version 1 files because the hash file code contains known bugs that can corrupt your data.
How do I generate random numbers in Python?

The standard module `random` implements a random number generator. Usage is simple:

```python
import random
random.random()
```

This returns a random floating point number in the range [0, 1).

There are also many other specialized generators in this module, such as:

- `randrange(a, b)` chooses an integer in the range [a, b).
- `uniform(a, b)` chooses a floating point number in the range [a, b).
- `normalvariate(mean, sdev)` samples the normal (Gaussian) distribution.

Some higher-level functions operate on sequences directly, such as:

- `choice(S)` chooses random element from a given sequence
- `shuffle(L)` shuffles a list in-place, i.e. permutes it randomly

There’s also a `Random` class you can instantiate to create independent multiple random number generators.
Extending/Embedding FAQ

Contents

- Extending/Embedding FAQ
  - Can I create my own functions in C?
  - Can I create my own functions in C++?
  - Writing C is hard; are there any alternatives?
  - How can I execute arbitrary Python statements from C?
  - How can I evaluate an arbitrary Python expression from C?
  - How do I extract C values from a Python object?
  - How do I use Py_BuildValue() to create a tuple of arbitrary length?
  - How do I call an object’s method from C?
  - How do I catch the output from PyErr_Print() (or anything that prints to stdout/stderr)?
  - How do I access a module written in Python from C?
  - How do I interface to C++ objects from Python?
  - I added a module using the Setup file and the make fails; why?
  - How do I debug an extension?
  - I want to compile a Python module on my Linux system, but some files are missing. Why?
  - What does “SystemError: _PyImport_FixupExtension: module yourmodule not loaded” mean?
  - How do I tell “incomplete input” from “invalid input”?  
  - How do I find undefined g++ symbols __builtin_new or __pure_virtual?
  - Can I create an object class with some methods implemented in C and others in Python (e.g. through inheritance)?
  - When importing module X, why do I get “undefined
symbol: PyUnicodeUCS2*""?
Can I create my own functions in C?

Yes, you can create built-in modules containing functions, variables, exceptions and even new types in C. This is explained in the document Extending and Embedding the Python Interpreter.

Most intermediate or advanced Python books will also cover this topic.
Can I create my own functions in C++?

Yes, using the C compatibility features found in C++. Place `extern "C" { ... }` around the Python include files and put `extern "C"` before each function that is going to be called by the Python interpreter. Global or static C++ objects with constructors are probably not a good idea.
Writing C is hard; are there any alternatives?

There are a number of alternatives to writing your own C extensions, depending on what you're trying to do.

If you need more speed, Psyco generates x86 assembly code from Python bytecode. You can use Psyco to compile the most time-critical functions in your code, and gain a significant improvement with very little effort, as long as you’re running on a machine with an x86-compatible processor.

Cython and its relative Pyrex are compilers that accept a slightly modified form of Python and generate the corresponding C code. Pyrex makes it possible to write an extension without having to learn Python’s C API.

If you need to interface to some C or C++ library for which no Python extension currently exists, you can try wrapping the library’s data types and functions with a tool such as SWIG. SIP, CXX Boost, or Weave are also alternatives for wrapping C++ libraries.
How can I execute arbitrary Python statements from C?

The highest-level function to do this is `PyRun_SimpleString()` which takes a single string argument to be executed in the context of the module `__main__` and returns 0 for success and -1 when an exception occurred (including `SyntaxError`). If you want more control, use `PyRun_String()`; see the source for `PyRun_SimpleString()` in Python/pythonrun.c.
How can I evaluate an arbitrary Python expression from C?

Call the function `PyRun_String()` from the previous question with the start symbol `Py_eval_input`; it parses an expression, evaluates it and returns its value.
How do I extract C values from a Python object?

That depends on the object's type. If it's a tuple, `PyTuple_Size()` returns its length and `PyTuple_GetItem()` returns the item at a specified index. Lists have similar functions, `PyListSize()` and `PyList_GetItem()`.

For strings, `PyString_Size()` returns its length and `PyString_AsString()` a pointer to its value. Note that Python strings may contain null bytes so C's `strlen()` should not be used.

To test the type of an object, first make sure it isn't `NULL`, and then use `PyString_Check()`, `PyTuple_Check()`, `PyList_Check()`, etc.

There is also a high-level API to Python objects which is provided by the so-called 'abstract' interface – read `include/abstract.h` for further details. It allows interfacing with any kind of Python sequence using calls like `PySequence_Length()`, `PySequence_GetItem()`, etc.) as well as many other useful protocols.
How do I use Py_BuildValue() to create a tuple of arbitrary length?

You can’t. Use \( t = \text{PyTuple\_New}(n) \) instead, and fill it with objects using \( \text{PyTuple\_SetItem}(t, i, o) \) – note that this “eats” a reference count of \( o \), so you have to \( \text{Py\_INCREF}() \) it. Lists have similar functions \( \text{PyList\_New}(n) \) and \( \text{PyList\_SetItem}(l, i, o) \). Note that you \textit{must} set all the tuple items to some value before you pass the tuple to Python code – \( \text{PyTuple\_New}(n) \) initializes them to NULL, which isn’t a valid Python value.
How do I call an object’s method from C?

The `PyObject_CallMethod()` function can be used to call an arbitrary method of an object. The parameters are the object, the name of the method to call, a format string like that used with `Py_BuildValue()`, and the argument values:

```c
PyObject *
PyObject_CallMethod(PyObject *object, char *method_name,
        char *arg_format, ...);
```

This works for any object that has methods – whether built-in or user-defined. You are responsible for eventually `Py_DECREF()`'ing the return value.

To call, e.g., a file object’s “seek” method with arguments 10, 0 (assuming the file object pointer is “f”):

```c
res = PyObject_CallMethod(f, "seek", "(ii)", 10, 0);
if (res == NULL) {
    ... an exception occurred ...
} else {
    Py_DECREF(res);
}
```

Note that since `PyObject_CallObject()` always wants a tuple for the argument list, to call a function without arguments, pass “()” for the format, and to call a function with one argument, surround the argument in parentheses, e.g. “(i)”. 
How do I catch the output from PyErr_Print() (or anything that prints to stdout/stderr)?

In Python code, define an object that supports the `write()` method. Assign this object to `sys.stdout` and `sys.stderr`. Call print_error, or just allow the standard traceback mechanism to work. Then, the output will go wherever your `write()` method sends it.

The easiest way to do this is to use the StringIO class in the standard library.

Sample code and use for catching stdout:

```python
>>> class StdoutCatcher:
...     def __init__(self):
...         self.data = ''
...     def write(self, stuff):
...         self.data = self.data + stuff

>>> import sys
>>> sys.stdout = StdoutCatcher()
>>> print 'foo'
>>> print 'hello world!'
>>> sys.stderr.write(sys.stdout.data)
foo
hello world!
```
How do I access a module written in Python from C?

You can get a pointer to the module object as follows:

\[
\text{module} = \text{PyImport_ImportModule}("<\text{modulename}>"); \]

If the module hasn't been imported yet (i.e. it is not yet present in \texttt{sys.modules}), this initializes the module; otherwise it simply returns the value of \texttt{sys.modules["<\text{modulename}>"]}. Note that it doesn't enter the module into any namespace – it only ensures it has been initialized and is stored in \texttt{sys.modules}.

You can then access the module's attributes (i.e. any name defined in the module) as follows:

\[
\text{attr} = \text{PyObject_GetAttrString}(<\text{module}, "<\text{attrname}>"}); \]

Calling \texttt{PyObject_SetAttrString()} to assign to variables in the module also works.
How do I interface to C++ objects from Python?

Depending on your requirements, there are many approaches. To do this manually, begin by reading the “Extending and Embedding” document. Realize that for the Python run-time system, there isn’t a whole lot of difference between C and C++ – so the strategy of building a new Python type around a C structure (pointer) type will also work for C++ objects.

For C++ libraries, see Writing C is hard; are there any alternatives?
I added a module using the Setup file and the make fails; why?

Setup must end in a newline, if there is no newline there, the build process fails. (Fixing this requires some ugly shell script hackery, and this bug is so minor that it doesn't seem worth the effort.)
How do I debug an extension?

When using GDB with dynamically loaded extensions, you can’t set a breakpoint in your extension until your extension is loaded.

In your `.gdbinit` file (or interactively), add the command:

```
br _PyImport_LoadDynamicModule
```

Then, when you run GDB:

```
$ gdb /local/bin/python
(gdb) run myscript.py
(gdb) continue # repeat until your extension is loaded
(gdb) finish     # so that your extension is loaded
(gdb) br myfunction.c:50
(gdb) continue
```
I want to compile a Python module on my Linux system, but some files are missing. Why?

Most packaged versions of Python don’t include the /usr/lib/python2.x/config/ directory, which contains various files required for compiling Python extensions.

For Red Hat, install the python-devel RPM to get the necessary files.

For Debian, run `apt-get install python-dev`. 
What does “SystemError: _PyImport_FixupExtension: module yourmodule not loaded” mean?

This means that you have created an extension module named “yourmodule”, but your module init function does not initialize with that name.

Every module init function will have a line similar to:

```python
module = Py_InitModule("yourmodule", yourmodule_functions);
```

If the string passed to this function is not the same name as your extension module, the `SystemError` exception will be raised.
How do I tell “incomplete input” from “invalid input”?

Sometimes you want to emulate the Python interactive interpreter’s behavior, where it gives you a continuation prompt when the input is incomplete (e.g. you typed the start of an “if” statement or you didn’t close your parentheses or triple string quotes), but it gives you a syntax error message immediately when the input is invalid.

In Python you can use the codeop module, which approximates the parser’s behavior sufficiently. IDLE uses this, for example.

The easiest way to do it in C is to call PyRun_InteractiveLoop() (perhaps in a separate thread) and let the Python interpreter handle the input for you. You can also set the PyOS_ReadlineFunctionPointer() to point at your custom input function. See Modules/readline.c and Parser/myreadline.c for more hints.

However sometimes you have to run the embedded Python interpreter in the same thread as your rest application and you can’t allow the PyRun_InteractiveLoop() to stop while waiting for user input. The one solution then is to call PyParser_ParseString() and test for e.error equal to E_EOF, which means the input is incomplete). Here’s a sample code fragment, untested, inspired by code from Alex Farber:

```c
#include <Python.h>
#include <node.h>
#include <errcode.h>
#include <grammar.h>
#include <parsetok.h>
#include <compile.h>
```
int testcomplete(char *code)  
/* code should end in \n */  
/* return -1 for error, 0 for incomplete, 1 for complete */  
{
    node *n;
    perrdetail e;

    n = PyParser_ParseString(code, &_PyParser_Grammar, Py_file_input, &e);
    if (n == NULL) {
        if (e.error == E_EOF)
            return 0;
        return -1;
    }

    PyNode_Free(n);
    return 1;
}

Another solution is trying to compile the received string with Py_CompileString(). If it compiles without errors, try to execute the returned code object by calling PyEval_EvalCode(). Otherwise save the input for later. If the compilation fails, find out if it’s an error or just more input is required - by extracting the message string from the exception tuple and comparing it to the string “unexpected EOF while parsing”. Here is a complete example using the GNU readline library (you may want to ignore SIGINT while calling readline()):

#include <stdio.h>
#include <readline.h>

#include <Python.h>
#include <object.h>
#include <compile.h>
#include <eval.h>

int main(int argc, char* argv[])
{
    int i, j, done = 0;  /* lengths of line */
    char ps1[] = ">>> ";
    char ps2[] = "... ";
    char *prompt = ps1;
    char *msg, *line, *code = NULL;

PyObject *src, *glb, *loc;

Py_Initialize();
loc = PyDict_New();
glb = PyDict_New();
PyDict_SetItemString (glb, "__builtins__", PyEval_GetBuiltins());

while (!done) {
    line = readline (prompt);

    if (NULL == line) /* Ctrl-D pressed */
        { done = 1; }
    else
    {
        i = strlen (line);

        if (i > 0)
            add_history (line); /* save non-empty lines */

        if (NULL == code) /* nothing in code yet */
            j = 0;
        else
            j = strlen (code);

        code = realloc (code, i + j + 2);

        if (NULL == code) /* out of memory */
            exit (1);

        if (0 == j) /* code was empty, so keep \0 */
            code[0] = '\0';

        strncat (code, line, i); /* append line to code */
        code[i + j] = '\n'; /* append \n to code */
        code[i + j + 1] = '\0';

        src = Py_CompileString (code, "<stdin>", Py_single_input);
        if (NULL != src) /* compiled just fine */
            {
                if (ps1 == prompt || /* ">>> " or */
                    '\n' == code[i + j - 1]) /* "... " and do */
                    /* so eval */
                    dum = PyEval_EvalCode ((PyCodeObject *)src, glb, loc);
Py_XDECREF (dum);
Py_XDECREF (src);
free (code);
    code = NULL;
    if (PyErr_Occurred ())
        PyErr_Print ();
    prompt = ps1;
}
    /* syntax error */
else if (PyErr_ExceptionMatches (PyExc_SyntaxError))
{
    PyErr_Fetch (&exc, &val, &trb);  /* clears exception! */

    if (PyArg_ParseTuple (val, "sO", &msg, &obj) &&
            !strcmp (msg, "unexpected EOF while parsing")) /* E_EOF */
    {
        Py_XDECREF (exc);
        Py_XDECREF (val);
        Py_XDECREF (trb);
        prompt = ps2;
    }
else  /* some other syntax error */
    {
        PyErr_Restore (exc, val, trb);
        PyErr_Print ();
        free (code);
        code = NULL;
        prompt = ps1;
    }
}
else  /* some non-syntax errors */
{
    PyErr_Print ();
    free (code);
    code = NULL;
    prompt = ps1;
}
free (line);
}

Py_XDECREF(glb);
Py_XDECREF(loc);
Py_Finalize();
exit(0);
How do I find undefined g++ symbols \_\_builtin\_new or \_\_pure\_virtual?

To dynamically load g++ extension modules, you must recompile Python, relink it using g++ (change LINKCC in the Python Modules Makefile), and link your extension module using g++ (e.g., `g++ -shared -o mymodule.so mymodule.o`).
Can I create an object class with some methods implemented in C and others in Python (e.g. through inheritance)?

Yes, you can inherit from built-in classes such as `int`, `list`, `dict`, etc.

The Boost Python Library (BPL, http://www.boost.org/libs/python/doc/index.html) provides a way of doing this from C++ (i.e. you can inherit from an extension class written in C++ using the BPL).
When importing module X, why do I get “undefined symbol: PyUnicodeUCS2*”?  

You are using a version of Python that uses a 4-byte representation for Unicode characters, but some C extension module you are importing was compiled using a Python that uses a 2-byte representation for Unicode characters (the default).

If instead the name of the undefined symbol starts with `PyUnicodeUCS4`, the problem is the reverse: Python was built using 2-byte Unicode characters, and the extension module was compiled using a Python with 4-byte Unicode characters.

This can easily occur when using pre-built extension packages. RedHat Linux 7.x, in particular, provided a “python2” binary that is compiled with 4-byte Unicode. This only causes the link failure if the extension uses any of the `PyUnicode_*()` functions. It is also a problem if an extension uses any of the Unicode-related format specifiers for `Py_BuildValue()` (or similar) or parameter specifications for `PyArg_ParseTuple()`.

You can check the size of the Unicode character a Python interpreter is using by checking the value of `sys.maxunicode`:

```python
>>> import sys
>>> if sys.maxunicode > 65535:
...     print 'UCS4 build'
... else:
...     print 'UCS2 build'
```

The only way to solve this problem is to use extension modules compiled with a Python binary built using the same size for Unicode characters.
Python on Windows FAQ

Contents

- Python on Windows FAQ
  - How do I run a Python program under Windows?
  - How do I make Python scripts executable?
  - Why does Python sometimes take so long to start?
  - How do I make an executable from a Python script?
  - Is a *.
pyd file the same as a DLL?
  - How can I embed Python into a Windows application?
  - How do I keep editors from inserting tabs into my Python source?
  - How do I check for a keypress without blocking?
  - How do I emulate os.kill() in Windows?
  - How do I extract the downloaded documentation on Windows?
How do I run a Python program under Windows?

This is not necessarily a straightforward question. If you are already familiar with running programs from the Windows command line then everything will seem obvious; otherwise, you might need a little more guidance.

Unless you use some sort of integrated development environment, you will end up typing Windows commands into what is variously referred to as a “DOS window” or “Command prompt window”. Usually you can create such a window from your Start menu; under Windows 7 the menu selection is Start ▶ Programs ▶ Accessories ▶ Command Prompt. You should be able to recognize when you have started such a window because you will see a Windows “command prompt”, which usually looks like this:

C:>

The letter may be different, and there might be other things after it, so you might just as easily see something like:

D:\YourName\Projects\Python>

depending on how your computer has been set up and what else
you have recently done with it. Once you have started such a window, you are well on the way to running Python programs.

You need to realize that your Python scripts have to be processed by another program called the Python interpreter. The interpreter reads your script, compiles it into bytecodes, and then executes the bytecodes to run your program. So, how do you arrange for the interpreter to handle your Python?

First, you need to make sure that your command window recognises the word “python” as an instruction to start the interpreter. If you have opened a command window, you should try entering the command python and hitting return.:

C:\Users\YourName> python

You should then see something like:

Python 2.7.3 (default, Apr 10 2012, 22:71:26) [MSC v.1500 32 bi
Type "help", "copyright", "credits" or "license" for more infor
>>>

You have started the interpreter in “interactive mode”. That means you can enter Python statements or expressions interactively and have them executed or evaluated while you wait. This is one of Python’s strongest features. Check it by entering a few expressions of your choice and seeing the results:

>>> print "Hello"
Hello
>>> "Hello" * 3
'HelloHelloHello'

Many people use the interactive mode as a convenient yet highly programmable calculator. When you want to end your interactive Python session, hold the ctrl key down while you enter a z, then hit
the “Enter” key to get back to your Windows command prompt.

You may also find that you have a Start-menu entry such as Start ▶ Programs ▶ Python 2.7 ▶ Python (command line) that results in you seeing the >>> prompt in a new window. If so, the window will disappear after you enter the ctrl-z character; Windows is running a single “python” command in the window, and closes it when you terminate the interpreter.

If the python command, instead of displaying the interpreter prompt >>>, gives you a message like:

```
'python' is not recognized as an internal or external command,
```

or:

```
Bad command or filename
```

then you need to make sure that your computer knows where to find the Python interpreter. To do this you will have to modify a setting called PATH, which is a list of directories where Windows will look for programs.

You should arrange for Python's installation directory to be added to the PATH of every command window as it starts. If you installed Python fairly recently then the command

```
dir C:\py*
```
will probably tell you where it is installed; the usual location is something like \texttt{C:\Python27}. Otherwise you will be reduced to a search of your whole disk ... use Tools $\Rightarrow$ Find or hit the Search button and look for “python.exe”. Supposing you discover that Python is installed in the \texttt{C:\Python27} directory (the default at the time of writing), you should make sure that entering the command

\begin{verbatim}
C:\Python27\python
\end{verbatim}

starts up the interpreter as above (and don’t forget you’ll need a “\texttt{Ctrl-Z}” and an “\texttt{Enter}” to get out of it). Once you have verified the directory, you can add it to the system path to make it easier to start Python by just running the \texttt{python} command. This is currently an option in the installer as of CPython 2.7.

More information about environment variables can be found on the \texttt{Using Python on Windows} page.
How do I make Python scripts executable?

On Windows, the standard Python installer already associates the .py extension with a file type (Python.File) and gives that file type an open command that runs the interpreter (D:\Program Files\Python\python.exe "%1" %*). This is enough to make scripts executable from the command prompt as ‘foo.py’. If you’d rather be able to execute the script by simple typing ‘foo’ with no extension you need to add .py to the PATHEXT environment variable.
Why does Python sometimes take so long to start?

Usually Python starts very quickly on Windows, but occasionally there are bug reports that Python suddenly begins to take a long time to start up. This is made even more puzzling because Python will work fine on other Windows systems which appear to be configured identically.

The problem may be caused by a misconfiguration of virus checking software on the problem machine. Some virus scanners have been known to introduce startup overhead of two orders of magnitude when the scanner is configured to monitor all reads from the filesystem. Try checking the configuration of virus scanning software on your systems to ensure that they are indeed configured identically. McAfee, when configured to scan all file system read activity, is a particular offender.
How do I make an executable from a Python script?

See [http://www.py2exe.org/](http://www.py2exe.org/) for a distutils extension that allows you to create console and GUI executables from Python code.
Is a *.pyd file the same as a DLL?

Yes, .pyd files are dll’s, but there are a few differences. If you have a DLL named foo.pyd, then it must have a function initfoo(). You can then write Python “import foo”, and Python will search for foo.pyd (as well as foo.py, foo.pyc) and if it finds it, will attempt to call initfoo() to initialize it. You do not link your .exe with foo.lib, as that would cause Windows to require the DLL to be present.

Note that the search path for foo.pyd is PYTHONPATH, not the same as the path that Windows uses to search for foo.dll. Also, foo.pyd need not be present to run your program, whereas if you linked your program with a dll, the dll is required. Of course, foo.pyd is required if you want to say import foo. In a DLL, linkage is declared in the source code with __declspec(dllexport). In a .pyd, linkage is defined in a list of available functions.
How can I embed Python into a Windows application?

Embedding the Python interpreter in a Windows app can be summarized as follows:

1. Do _not_ build Python into your .exe file directly. On Windows, Python must be a DLL to handle importing modules that are themselves DLL's. (This is the first key undocumented fact.) Instead, link to `pythonNN.dll`; it is typically installed in `C:\Windows\System`. `NN` is the Python version, a number such as “27” for Python 2.7.

   You can link to Python in two different ways. Load-time linking means linking against `pythonNN.lib`, while run-time linking means linking against `pythonNN.dll`. (General note: `pythonNN.lib` is the so-called “import lib” corresponding to `pythonNN.dll`. It merely defines symbols for the linker.)

   Run-time linking greatly simplifies link options; everything happens at run time. Your code must load `pythonNN.dll` using the Windows `LoadLibraryEx()` routine. The code must also use access routines and data in `pythonNN.dll` (that is, Python's C API's) using pointers obtained by the Windows `GetProcAddress()` routine. Macros can make using these pointers transparent to any C code that calls routines in Python's C API.

   Borland note: convert `pythonNN.lib` to OMF format using `Coff2Omf.exe` first.

2. If you use SWIG, it is easy to create a Python “extension
module” that will make the app’s data and methods available to Python. SWIG will handle just about all the grungy details for you. The result is C code that you link into your .exe file (!) You do _not_ have to create a DLL file, and this also simplifies linking.

3. SWIG will create an init function (a C function) whose name depends on the name of the extension module. For example, if the name of the module is leo, the init function will be called initleo(). If you use SWIG shadow classes, as you should, the init function will be called initleoc(). This initializes a mostly hidden helper class used by the shadow class.

The reason you can link the C code in step 2 into your .exe file is that calling the initialization function is equivalent to importing the module into Python! (This is the second key undocumented fact.)

4. In short, you can use the following code to initialize the Python interpreter with your extension module.

```c
#include "python.h"
...
Py_Initialize(); // Initialize Python.
initmyAppc(); // Initialize (import) the helper class.
PyRun_SimpleString("import myApp"); // Import the shadow class.
```

5. There are two problems with Python’s C API which will become apparent if you use a compiler other than MSVC, the compiler used to build pythonNN.dll.

Problem 1: The so-called “Very High Level” functions that take FILE * arguments will not work in a multi-compiler environment because each compiler’s notion of a struct FILE will be different. From an implementation standpoint these are very _low_ level functions.
Problem 2: SWIG generates the following code when generating wrappers to void functions:

```c
Py_INCREF(Py_None);
_resultobj = Py_None;
return _resultobj;
```

Alas, Py_None is a macro that expands to a reference to a complex data structure called _Py_NoneStruct inside pythonNN.dll. Again, this code will fail in a mult-compiler environment. Replace such code by:

```c
return Py_BuildValue(" ");
```

It may be possible to use SWIG's `%typemap` command to make the change automatically, though I have not been able to get this to work (I'm a complete SWIG newbie).

6. Using a Python shell script to put up a Python interpreter window from inside your Windows app is not a good idea; the resulting window will be independent of your app's windowing system. Rather, you (or the wxPythonWindow class) should create a “native” interpreter window. It is easy to connect that window to the Python interpreter. You can redirect Python's i/o to _any_ object that supports read and write, so all you need is a Python object (defined in your extension module) that contains read() and write() methods.
How do I keep editors from inserting tabs into my Python source?

The FAQ does not recommend using tabs, and the Python style guide, PEP 8, recommends 4 spaces for distributed Python code; this is also the Emacs python-mode default.

Under any editor, mixing tabs and spaces is a bad idea. MSVC is no different in this respect, and is easily configured to use spaces: Take Tools ▶ Options ▶ Tabs, and for file type “Default” set “Tab size” and “Indent size” to 4, and select the “Insert spaces” radio button.

If you suspect mixed tabs and spaces are causing problems in leading whitespace, run Python with the -t switch or run Tools/Scripts/tabnanny.py to check a directory tree in batch mode.
How do I check for a keypress without blocking?

Use the msvcrt module. This is a standard Windows-specific extension module. It defines a function `kbhit()` which checks whether a keyboard hit is present, and `getch()` which gets one character without echoing it.
How do I emulate os.kill() in Windows?

Prior to Python 2.7 and 3.2, to terminate a process, you can use `ctypes`:

```python
import ctypes

def kill(pid):
    """kill function for Win32""
    kernel32 = ctypes.windll.kernel32
    handle = kernel32.OpenProcess(1, 0, pid)
    return (0 != kernel32.TerminateProcess(handle, 0))
```

In 2.7 and 3.2, `os.kill()` is implemented similar to the above function, with the additional feature of being able to send `Ctrl+C` and `Ctrl+Break` to console subprocesses which are designed to handle those signals. See `os.kill()` for further details.
How do I extract the downloaded documentation on Windows?

Sometimes, when you download the documentation package to a Windows machine using a web browser, the file extension of the saved file ends up being .EXE. This is a mistake; the extension should be .TGZ.

Simply rename the downloaded file to have the .TGZ extension, and WinZip will be able to handle it. (If your copy of WinZip doesn’t, get a newer one from https://www.winzip.com.)
Graphic User Interface FAQ

Contents

- Graphic User Interface FAQ
  - What platform-independent GUI toolkits exist for Python?
    - Tkinter
    - wxWidgets
    - Qt
    - Gtk+
    - FLTK
    - FOX
    - OpenGL
  - What platform-specific GUI toolkits exist for Python?
  - Tkinter questions
    - How do I freeze Tkinter applications?
    - Can I have Tk events handled while waiting for I/O?
    - I can't get key bindings to work in Tkinter: why?
What platform-independent GUI toolkits exist for Python?

Depending on what platform(s) you are aiming at, there are several.

Tkinter

Standard builds of Python include an object-oriented interface to the Tcl/Tk widget set, called Tkinter. This is probably the easiest to install and use. For more info about Tk, including pointers to the source, see the Tcl/Tk home page at [https://www.tcl.tk](https://www.tcl.tk). Tcl/Tk is fully portable to the Mac OS X, Windows, and Unix platforms.

wxWidgets

wxWidgets ([https://www.wxwidgets.org](https://www.wxwidgets.org)) is a free, portable GUI class library written in C++ that provides a native look and feel on a number of platforms, with Windows, Mac OS X, GTK, X11, all listed as current stable targets. Language bindings are available for a number of languages including Python, Perl, Ruby, etc.

wxPython ([http://www.wxpython.org](http://www.wxpython.org)) is the Python binding for wxwidgets. While it often lags slightly behind the official wxWidgets releases, it also offers a number of features via pure Python extensions that are not available in other language bindings. There is an active wxPython user and developer community.

Both wxWidgets and wxPython are free, open source, software with permissive licences that allow their use in commercial products as well as in freeware or shareware.

Qt
There are bindings available for the Qt toolkit (using either PyQt or PySide) and for KDE (PyKDE4). PyQt is currently more mature than PySide, but you must buy a PyQt license from Riverbank Computing if you want to write proprietary applications. PySide is free for all applications.

Qt 4.5 upwards is licensed under the LGPL license; also, commercial licenses are available from The Qt Company.

**Gtk+**

PyGtk bindings for the Gtk+ toolkit have been implemented by James Henstridge; see <http://www.pygtk.org>.

**FLTK**

Python bindings for the FLTK toolkit, a simple yet powerful and mature cross-platform windowing system, are available from the PyFLTK project.

**FOX**

A wrapper for the FOX toolkit called FXpy is available. FOX supports both Unix variants and Windows.

**OpenGL**

For OpenGL bindings, see PyOpenGL.
What platform-specific GUI toolkits exist for Python?

By installing the PyObjc Objective-C bridge, Python programs can use Mac OS X’s Cocoa libraries.

Pythonwin by Mark Hammond includes an interface to the Microsoft Foundation Classes and a Python programming environment that’s written mostly in Python using the MFC classes.
Tkinter questions

How do I freeze Tkinter applications?

Freeze is a tool to create stand-alone applications. When freezing Tkinter applications, the applications will not be truly stand-alone, as the application will still need the Tcl and Tk libraries.

One solution is to ship the application with the Tcl and Tk libraries, and point to them at run-time using the `TCL_LIBRARY` and `TK_LIBRARY` environment variables.

To get truly stand-alone applications, the Tcl scripts that form the library have to be integrated into the application as well. One tool supporting that is SAM (stand-alone modules), which is part of the Tix distribution (http://tix.sourceforge.net/).

Build Tix with SAM enabled, perform the appropriate call to `Tclsam_init()`, etc. inside Python’s `Modules/tkappinit.c`, and link with `libtclsam` and `libtksam` (you might include the Tix libraries as well).

Can I have Tk events handled while waiting for I/O?

On platforms other than Windows, yes, and you don't even need threads! But you'll have to restructure your I/O code a bit. Tk has the equivalent of Xt’s `XtAddInput()` call, which allows you to register a callback function which will be called from the Tk mainloop when I/O is possible on a file descriptor. See File Handlers.

I can’t get key bindings to work in Tkinter: why?

An often-heard complaint is that event handlers bound to events with
the `bind()` method don’t get handled even when the appropriate key is pressed.

The most common cause is that the widget to which the binding applies doesn’t have “keyboard focus”. Check out the Tk documentation for the focus command. Usually a widget is given the keyboard focus by clicking in it (but not for labels; see the takefocus option).
“Why is Python Installed on my Computer?” FAQ
What is Python?

Python is a programming language. It’s used for many different applications. It’s used in some high schools and colleges as an introductory programming language because Python is easy to learn, but it’s also used by professional software developers at places such as Google, NASA, and Lucasfilm Ltd.

If you wish to learn more about Python, start with the Beginner's Guide to Python.
Why is Python installed on my machine?

If you find Python installed on your system but don't remember installing it, there are several possible ways it could have gotten there.

- Perhaps another user on the computer wanted to learn programming and installed it; you'll have to figure out who's been using the machine and might have installed it.
- A third-party application installed on the machine might have been written in Python and included a Python installation. For a home computer, the most common such application is PySol, a solitaire game that includes over 1000 different games and variations.
- Some Windows machines also have Python installed. At this writing we're aware of computers from Hewlett-Packard and Compaq that include Python. Apparently some of HP/Compaq's administrative tools are written in Python.
- All Apple computers running Mac OS X have Python installed; it's included in the base installation.
Can I delete Python?

That depends on where Python came from.

If someone installed it deliberately, you can remove it without hurting anything. On Windows, use the Add/Remove Programs icon in the Control Panel.

If Python was installed by a third-party application, you can also remove it, but that application will no longer work. You should use that application’s uninstaller rather than removing Python directly.

If Python came with your operating system, removing it is not recommended. If you remove it, whatever tools were written in Python will no longer run, and some of them might be important to you. Reinstalling the whole system would then be required to fix things again.
The default Python prompt of the interactive shell. Often seen for code examples which can be executed interactively in the interpreter.

The default Python prompt of the interactive shell when entering code for an indented code block or within a pair of matching left and right delimiters (parentheses, square brackets or curly braces).

2to3
A tool that tries to convert Python 2.x code to Python 3.x code by handling most of the incompatibilities which can be detected by parsing the source and traversing the parse tree.

2to3 is available in the standard library as `lib2to3`; a standalone entry point is provided as `Tools/scripts/2to3`. See 2to3 - Automated Python 2 to 3 code translation.

**abstract base class**
Abstract base classes complement duck-typing by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don’t inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections` module), numbers (in the `numbers` module), and streams (in the `io` module). You can create your own ABCs with the `abc` module.

**argument**
A value passed to a function (or method) when calling the function. There are two types of arguments:

- **keyword argument**: an argument preceded by an identifier (e.g. `name=`) in a function call or passed as a value in a dictionary preceded by `**`. For example, 3 and 5 are both keyword arguments in the following calls to `complex()`:

  ```python
  complex(real=3, imag=5)
  complex(**{'real': 3, 'imag': 5})
  ```

- **positional argument**: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an iterable preceded by `*`. For example, 3 and 5 are both positional arguments in the following calls:

  ```python
  complex(3, 5)
  complex(*[3, 5])
  ```

Arguments are assigned to the named local variables in a function body. See the Calls section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

See also the parameter glossary entry and the FAQ question on the difference between arguments and parameters.

**attribute**

A value associated with an object which is referenced by name using dotted expressions. For example, if an object `o` has an attribute `a` it would be referenced as `o.a`.

**BDFL**

Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's
creator.

**bytes-like object**

An object that supports the buffer protocol, like `str`, `bytearray` or `memoryview`. Bytes-like objects can be used for various operations that expect binary data, such as compression, saving to a binary file or sending over a socket. Some operations need the binary data to be mutable, in which case not all bytes-like objects can apply.

**bytecode**

Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This “intermediate language” is said to run on a virtual machine that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

A list of bytecode instructions can be found in the documentation for the `dis` module.

**class**

A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

**classic class**

Any class which does not inherit from `object`. See new-style class. Classic classes have been removed in Python 3.

**coercion**

The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same
type. For example, \texttt{int(3.15)} converts the floating point number to the integer 3, but in \texttt{3+4.5}, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a \texttt{TypeError}. Coercion between two operands can be performed with the \texttt{coerce} built-in function; thus, \texttt{3+4.5} is equivalent to calling \texttt{operator.add(*coerce(3, 4.5))} and results in \texttt{operator.add(3.0, 4.5)}. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., \texttt{float(3)+4.5} rather than just \texttt{3+4.5}.

**complex number**

An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of \(-1\)), often written \texttt{i} in mathematics or \texttt{j} in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a \texttt{j} suffix, e.g., \texttt{3+1j}. To get access to complex equivalents of the \texttt{math} module, use \texttt{cmath}. Use of complex numbers is a fairly advanced mathematical feature. If you’re not aware of a need for them, it’s almost certain you can safely ignore them.

**context manager**

An object which controls the environment seen in a \texttt{with} statement by defining \texttt{__enter__()} and \texttt{__exit__()} methods. See \texttt{PEP 343}.

**CPython**

The canonical implementation of the Python programming language, as distributed on \texttt{python.org}. The term “CPython” is used when necessary to distinguish this implementation from others such as Jython or IronPython.
**decorator**

A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for decorators are `classmethod()` and `staticmethod()`.

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```python
def f(...):
    ...
    f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for function definitions and class definitions for more about decorators.

**descriptor**

Any *new-style* object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using `a.b` to get, set or delete an attribute looks up the object named `b` in the class dictionary for `a`, but if `b` is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

For more information about descriptors’ methods, see Implementing Descriptors.

**dictionary**
An associative array, where arbitrary keys are mapped to values. The keys can be any object with \_\_hash\_() and \_\_eq\_() methods. Called a hash in Perl.

dictionary view
The objects returned from \_\_viewkeys\_(), \_\_viewvalues\_(), and \_\_viewitems\_() are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use \_\_list\_(). See Dictionary view objects.

docstring
A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the \_\_doc\_ attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

duck-typing
A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used ("If it looks like a duck and quacks like a duck, it must be a duck."") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using \_\_type\_() or \_\_instance\_(). (Note, however, that duck-typing can be complemented with abstract base classes.) Instead, it typically employs \_\_hasattr\_() tests or EAFP programming.

EAFP
Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or
attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many try and except statements. The technique contrasts with the LBYL style common to many other languages such as C.

**expression**
A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also statements which cannot be used as expressions, such as print or if. Assignments are also statements, not expressions.

**extension module**
A module written in C or C++, using Python’s C API to interact with the core and with user code.

**file object**
An object exposing a file-oriented API (with methods such as read() or write()) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called file-like objects or streams.

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the io module. The canonical way to create a file object is by using the open() function.

**file-like object**
A synonym for file object.

**finder**
An object that tries to find the loader for a module. It must implement a method named find_module(). See PEP 302 for details.

floor division
Mathematical division that rounds down to nearest integer. The floor division operator is //. For example, the expression 11 // 4 evaluates to 2 in contrast to the 2.75 returned by float true division. Note that (-11) // 4 is -3 because that is -2.75 rounded downward. See PEP 238.

function
A series of statements which returns some value to a caller. It can also be passed zero or more arguments which may be used in the execution of the body. See also parameter, method, and the Function definitions section.

__future__
A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression 11/4 currently evaluates to 2. If the module in which it is executed had enabled true division by executing:

```python
from __future__ import division
```

the expression 11/4 would evaluate to 2.75. By importing the __future__ module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default:

```python
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```
**garbage collection**

The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles.

**generator**

A function which returns an iterator. It looks like a normal function except that it contains `yield` statements for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function. Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks-up where it left-off (in contrast to functions which start fresh on every invocation).

**generator expression**

An expression that returns an iterator. It looks like a normal expression followed by a `for` expression defining a loop variable, range, and an optional `if` expression. The combined expression generates values for an enclosing function:

```python
>>> sum(i*i for i in range(10))  # sum of squares 0, 1, 4, ..., 81
285
```

**GIL**

See global interpreter lock.

**global interpreter lock**

The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as `dict`) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of
much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

Past efforts to create a “free-threaded” interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case. It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.

**hashable**

An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__() ` or `__cmp__() ` method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python’s immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal (except with themselves), and their hash value is derived from their `id()`.

**IDLE**

An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment which ships with the standard distribution of Python.
immutable
An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

integer division
Mathematical division discarding any remainder. For example, the expression \( \frac{11}{4} \) currently evaluates to 2 in contrast to the 2.75 returned by float division. Also called floor division. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a float), the result will be coerced (see coercion) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also `__future__`.

importing
The process by which Python code in one module is made available to Python code in another module.

importer
An object that both finds and loads a module; both a finder and loader object.

interactive
Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer’s main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`).

interpreted
Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also interactive.

**iterable**

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as list, str, and tuple) and some non-sequence types like dict and file and objects of any classes you define with an __iter__() or __getitem__() method. Iterables can be used in a for loop and in many other places where a sequence is needed (zip(), map(), ...). When an iterable object is passed as an argument to the built-in function iter(), it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call iter() or deal with iterator objects yourself. The for statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also iterator, sequence, and generator.

**iterator**

An object representing a stream of data. Repeated calls to the iterator’s next() method return successive items in the stream. When no more data are available a StopIteration exception is raised instead. At this point, the iterator object is exhausted and any further calls to its next() method just raise StopIteration again. Iterators are required to have an __iter__() method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple
iteration passes. A container object (such as a list) produces a fresh new iterator each time you pass it to the iter() function or use it in a for loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in Iterator Types.

**key function**

A key function or collation function is a callable that returns a value used for sorting or ordering. For example, locale.strxfrm() is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include min(), max(), sorted(), list.sort(), heapq.nsmallest(), heapq.nlargest(), and itertools.groupby().

There are several ways to create a key function. For example, the str.lower() method can serve as a key function for case insensitive sorts. Alternatively, an ad-hoc key function can be built from a lambda expression such as lambda r: (r[0], r[2]). Also, the operator module provides three key function constructors: attrgetter(), itemgetter(), and methodcaller(). See the Sorting HOW TO for examples of how to create and use key functions.

**keyword argument**

See argument.

**lambda**

An anonymous inline function consisting of a single expression which is evaluated when the function is called. The syntax to
create a lambda function is `lambda [arguments]: expression`

**LBYL**
Look before you leap. This coding style explicitly tests for preconditions before making calls or lookups. This style contrasts with the EAFP approach and is characterized by the presence of many `if` statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between “the looking” and “the leaping”. For example, the code, `if key in mapping: return mapping[key]` can fail if another thread removes `key` from `mapping` after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

**list**
A built-in Python sequence. Despite its name it is more akin to an array in other languages than to a linked list since access to elements are O(1).

**list comprehension**
A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

**loader**
An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a `finder`. See PEP 302 for details.

**mapping**
A container object that supports arbitrary key lookups and
implements the methods specified in the `Mapping` or `MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

**metaclass**

The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object-oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in **Customizing class creation**.

**method**

A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first argument (which is usually called `self`). See function and **nested scope**.

**method resolution order**

Method Resolution Order is the order in which base classes are searched for a member during lookup. See [The Python 2.3 Method Resolution Order](https://docs.python.org/3.8/c-api/class.html) for details of the algorithm used by the Python interpreter since the 2.3 release.

**module**

An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of **importing**.
See also package.

**MRO**

See method resolution order.

**mutable**

Mutable objects can change their value but keep their id(). See also immutable.

**named tuple**

Any tuple-like class whose indexable elements are also accessible using named attributes (for example, `time.localtime()` returns a tuple-like object where the year is accessible either with an index such as `t[0]` or with a named attribute like `t.tm_year`).

A named tuple can be a built-in type such as `time.struct_time`, or it can be created with a regular class definition. A full featured named tuple can also be created with the factory function `collections.namedtuple()`. The latter approach automatically provides extra features such as a self-documenting representation like `Employee(name='jones', title='programmer')`.

**namespace**

The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules,
respectively.

**nested scope**
The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

**new-style class**
Any class which inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python’s newer, versatile features like `__slots__`, descriptors, properties, and `__getattribute__()`. More information can be found in New-style and classic classes.

**object**
Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any new-style class.

**package**
A Python module which can contain submodules or recursively, subpackages. Technically, a package is a Python module with an `__path__` attribute.

**parameter**
A named entity in a function (or method) definition that specifies an argument (or in some cases, arguments) that the function can accept. There are four types of parameters:

- *positional-or-keyword*: specifies an argument that can be passed either **positionally** or as a **keyword argument**. This is the default kind of parameter, for example `foo` and `bar` in the
following:

```python
def func(foo, bar=None):
...
```

- **positional-only**: specifies an argument that can be supplied only by position. Python has no syntax for defining positional-only parameters. However, some built-in functions have positional-only parameters (e.g. `abs()`).

- **var-positional**: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `*`, for example `args` in the following:

```python
def func(*args, **kwargs):
...
```

- **var-keyword**: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `**`, for example `kwargs` in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the argument glossary entry, the FAQ question on the difference between arguments and parameters, and the Function definitions section.

**positional argument**

See argument.

**Python 3000**

Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This
is also abbreviated “Py3k”.

**Pythonic**

An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a `for` statement. Many other languages don’t have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```python
for i in range(len(food)):
    print food[i]
```

As opposed to the cleaner, Pythonic method:

```python
for piece in food:
    print piece
```

**reference count**

The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Reference counting is generally not visible to Python code, but it is a key element of the CPython implementation. The `sys` module defines a `getrefcount()` function that programmers can call to return the reference count for a particular object.

**__slots__**

A declaration inside a **new-style class** that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

**sequence**

An **iterable** which supports efficient element access using integer
indices via the __getitem__() special method and defines a len() method that returns the length of the sequence. Some built-in sequence types are list, str, tuple, and unicode. Note that dict also supports __getitem__() and __len__(), but is considered a mapping rather than a sequence because the lookups use arbitrary immutable keys rather than integers.

slice
An object usually containing a portion of a sequence. A slice is created using the subscript notation, [] with colons between numbers when several are given, such as in variable_name[1:3:5]. The bracket (subscript) notation uses slice objects internally (or in older versions, __getslice__() and __setslice__()).

special method
A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in Special method names.

statement
A statement is part of a suite (a “block” of code). A statement is either an expression or one of several constructs with a keyword, such as if, while or for.

struct sequence
A tuple with named elements. Struct sequences expose an interface similar to named tuple in that elements can either be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like _make() or _asdict(). Examples of struct sequences include sys.float_info and the return value of os.stat().

triple-quoted string
A string which is bound by three instances of either a quotation mark (" ) or an apostrophe ('). While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

**type**

The type of a Python object determines what kind of object it is; every object has a type. An object’s type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

**universal newlines**

A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `
`, the Windows convention `
`, and the old Macintosh convention ``. See [PEP 278](https://peps.python.org/pep-0278/) and [PEP 3116](https://peps.python.org/pep-03116/), as well as `str.splitlines()` for an additional use.

**virtual environment**

A cooperatively isolated runtime environment that allows Python users and applications to install and upgrade Python distribution packages without interfering with the behaviour of other Python applications running on the same system.

**virtual machine**

A computer defined entirely in software. Python’s virtual machine executes the bytecode emitted by the bytecode compiler.

**Zen of Python**

Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing “`import this`” at the interactive prompt.
About these documents

These documents are generated from reStructuredText sources by Sphinx, a document processor specifically written for the Python documentation.

Development of the documentation and its toolchain is an entirely volunteer effort, just like Python itself. If you want to contribute, please take a look at the Reporting Bugs page for information on how to do so. New volunteers are always welcome!

Many thanks go to:

- Fred L. Drake, Jr., the creator of the original Python documentation toolset and writer of much of the content;
- the Docutils project for creating reStructuredText and the Docutils suite;
- Fredrik Lundh for his Alternative Python Reference project from which Sphinx got many good ideas.
Many people have contributed to the Python language, the Python standard library, and the Python documentation. See Misc/ACKS in the Python source distribution for a partial list of contributors.

It is only with the input and contributions of the Python community that Python has such wonderful documentation – Thank You!
Reporting Bugs

Python is a mature programming language which has established a reputation for stability. In order to maintain this reputation, the developers would like to know of any deficiencies you find in Python.
Documentation bugs

If you find a bug in this documentation or would like to propose an improvement, please submit a bug report on the tracker. If you have a suggestion how to fix it, include that as well.

If you’re short on time, you can also email documentation bug reports to docs@python.org (behavioral bugs can be sent to python-list@python.org). ‘docs@’ is a mailing list run by volunteers; your request will be noticed, though it may take a while to be processed.

See also: Documentation bugs on the Python issue tracker
Using the Python issue tracker

Bug reports for Python itself should be submitted via the Python Bug Tracker (https://bugs.python.org/). The bug tracker offers a Web form which allows pertinent information to be entered and submitted to the developers.

The first step in filing a report is to determine whether the problem has already been reported. The advantage in doing so, aside from saving the developers time, is that you learn what has been done to fix it; it may be that the problem has already been fixed for the next release, or additional information is needed (in which case you are welcome to provide it if you can!). To do this, search the bug database using the search box on the top of the page.

If the problem you’re reporting is not already in the bug tracker, go back to the Python Bug Tracker and log in. If you don’t already have a tracker account, select the “Register” link or, if you use OpenID, one of the OpenID provider logos in the sidebar. It is not possible to submit a bug report anonymously.

Being now logged in, you can submit a bug. Select the “Create New” link in the sidebar to open the bug reporting form.

The submission form has a number of fields. For the “Title” field, enter a very short description of the problem; less than ten words is good. In the “Type” field, select the type of your problem; also select the “Component” and “Versions” to which the bug relates.

In the “Comment” field, describe the problem in detail, including what you expected to happen and what did happen. Be sure to include whether any extension modules were involved, and what hardware and software platform you were using (including version information as appropriate).
Each bug report will be assigned to a developer who will determine what needs to be done to correct the problem. You will receive an update each time action is taken on the bug.

See also:

**How to Report Bugs Effectively**
Article which goes into some detail about how to create a useful bug report. This describes what kind of information is useful and why it is useful.

**Bug Writing Guidelines**
Information about writing a good bug report. Some of this is specific to the Mozilla project, but describes general good practices.
Getting started contributing to Python yourself

Beyond just reporting bugs that you find, you are also welcome to submit patches to fix them. You can find more information on how to get started patching Python in the Python Developer’s Guide. If you have questions, the core-mentorship mailing list is a friendly place to get answers to any and all questions pertaining to the process of fixing issues in Python.
Copyright

Python and this documentation is:

Copyright © 2001-2016 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

See History and License for complete license and permissions information.
History and License
History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python’s principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

<table>
<thead>
<tr>
<th>Release</th>
<th>Derived from</th>
<th>Year</th>
<th>Owner</th>
<th>GPL compatible?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9.0 thru 1.2</td>
<td>n/a</td>
<td>1991-1995</td>
<td>CWI</td>
<td>yes</td>
</tr>
<tr>
<td>1.3 thru 1.5.2</td>
<td>1.2</td>
<td>1995-1999</td>
<td>CNRI</td>
<td>yes</td>
</tr>
<tr>
<td>1.6</td>
<td>1.5.2</td>
<td>2000</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>Version</td>
<td>From</td>
<td>Year</td>
<td>Distributor</td>
<td>GPL Compatible</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------</td>
<td>--------</td>
<td>-------------</td>
<td>----------------</td>
</tr>
<tr>
<td>2.0</td>
<td>1.6</td>
<td>2000</td>
<td>BeOpen.com</td>
<td>no</td>
</tr>
<tr>
<td>1.6.1</td>
<td>1.6</td>
<td>2001</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>no</td>
</tr>
<tr>
<td>2.0.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.1</td>
<td>2.1+2.0.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.2</td>
<td>2.1.1</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.3</td>
<td>2.1.2</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2 and above</td>
<td>2.1.1</td>
<td>2001-now</td>
<td>PSF</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Note:** GPL-compatible doesn’t mean that we’re distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don’t.

Thanks to the many outside volunteers who have worked under Guido’s direction to make these releases possible.
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.7.13 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.7.13 alone or in any version, provided, however, that PSF's License Agreement and copyright, i.e., "Copyright © 2001-2016 Python Software Foundation; All Rights Reserved" are retained in Python 2.7.13 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.7.13 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee agrees to include in any such work a brief summary of the changes made to Python 2.7.13.

4. PSF is making Python 2.7.13 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.7.13 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.13 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSSES RESULTING FROM THE USE OF PYTHON 2.7.13, OR OTHERWISE USING PYTHON 2.7.13 OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create a relationship of agency, partnership, or joint venture between PSF and Licensee.
Agreement does not grant permission to use PSF trademarks or trademark sense to endorse or promote products or services of a third party.

8. By copying, installing or otherwise using Python 2.7.13, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT
VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, to reproduce, analyze, test, perform and/or display publicly works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted by the law of the State of California, excluding conflict of Nothing in this License Agreement shall be deemed to create agency, partnership, or joint venture between BeOpen and Lic
Agreement does not grant permission to use BeOpen trademarks in a trademark sense to endorse or promote products or services of a third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice, i.e., "Copyright © 1995-2001 Corporation for National Research Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes) is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be obtained from the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013.

3. In the event Licensee prepares a derivative work that is based on Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" BASIS. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATIONS OR WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the Commonwealth of Virginia, excluding Virginia's conflict of law rules. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. This License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee or any third party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF SOFTWARE.
Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

Mersenne Twister

The `_random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

A C-program for MT19937, with initialization improved 2002/1/26
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using `init_genrand(seed)` or `init_by_array(init_key, key_length)`.  

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
Sockets

The socket module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/.

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS AS IS, ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE...
Floating point exception control

The source for the fpectl module includes the following notice:

---------------------------------------------------------------------
/  Copyright (c) 1996.
|  The Regents of the University of California.
|  All rights reserved.
|
|  Permission to use, copy, modify, and distribute this software
|  any purpose without fee is hereby granted, provided that the
|  entire notice is included in all copies of any software which
|  includes a copy or modification of this software and copies
|  of the supporting documentation for such software.
|
|  This work was produced at the University of California, Lawrence
|  Livermore National Laboratory under Contract No. W-7405-
|  between the U.S. Department of Energy and The Regents
|  of the University of California for the operation of UC LLNL.

DISCLAIMER

This software was prepared as an account of work sponsored
agency of the United States Government. Neither the United
Government nor the University of California nor any of its
employees, makes any warranty, express or implied, or assume
liability or responsibility for the accuracy, completeness
usefulness of any information, apparatus, product, or
information, apparatus, product, or process disclosed, or
represents that its use would not infringe privately-owned
rights. Reference herein to any specific
commercial products, process, or service by trade name, trad
manufacturer, or otherwise, does not necessarily constitute
imply its endorsement, recommendation, or favoring by the
States Government or the University of California. The views
opinions of authors expressed herein do not necessarily
reflect those of the United States Government or the University
of California, and shall not be used for advertising or
endorsement purposes.
MD5 message digest algorithm

The source code for the md5 module contains the following notice:

Copyright (C) 1999, 2002 Aladdin Enterprises. All rights reserved.

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any arising from the use of this software.

Permission is granted to anyone to use this software for any purpose including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you may not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and not misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose text is available at
    http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite (section A.5) but excluding the rest of Appendix A. It does not include any code or documentation that is identified in the RFC as being copyrighted.

The original and principal author of md5.h is L. Peter Deutsch <ghost@aladdin.com>. Other authors are noted in the change history that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed references to Ghostscript; clarified derivation from RFC now handles byte order either statically or dynamically.
Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Cookie management

The `cookie` module contains the following notice:

Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this software
Execution tracing

The `trace` module contains the following notice:

```plaintext
portions copyright 2001, Autonomous Zones Industries, Inc., all err... reserved and offered to the public under the terms of the Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights

Permission to use, copy, modify, and distribute this Python sof
its associated documentation for any purpose without fee is her
```
UUencode and UUdecode functions

The `uu` module contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
   All Rights Reserved
Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and
that the name of Lance Ellinghouse not be used in advertising or
publicity pertaining to distribution of the software without specific,
written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING
USED IN ADVERTISING OR PUBLICITY PERTAINING TO DISTRIBUTION
OF THE SOFTWARE WITHOUT SPECIFIC, WRITTEN PRIOR PERMISSION.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

XML Remote Procedure Calls

The `xmlrpcclib` module contains the following notice:

```
The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh
```
By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publication pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

test_epoll

The test_epoll contains the following notice:

Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
Select kqueue

The select and contains the following notice for the kqueue interface:

Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in
   the documentation and/or other materials provided with the
distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS AS IS AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

strtod and dtoa

The file Python/dtoa.c, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from http://www.netlib.org/fp/. The original file, as retrieved on March 16,
2009, contains the following copyright and licensing notice:

```c
****************************************************************
* The author of this software is David M. Gay.

* Copyright (c) 1991, 2000, 2001 by Lucent Technologies.

* Permission to use, copy, modify, and distribute this software
  without fee is hereby granted, provided that this entire
  notice is included in all copies of any software which is or
  includes or modification of this software and in all copies of
  the supporting documentation for such software.

* THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS
  WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKE
  REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANDISE
  OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

****************************************************************
```

OpenSSL

The modules `hashlib`, `posix`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and Mac OS X installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

**LICENSE ISSUES**

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of the OpenSSL License and the original SSLeay license apply to this software. See below for the actual license texts. Actually both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL please contact openssl-core@openssl.org.

OpenSSL License

--------------
Original SSLeay License
-----------------------

/*
Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
All rights reserved.
*/

* This package is an SSL implementation written
  by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netsc

* This library is free for commercial and non-commercial use
  the following conditions are aheared to. The following co
  apply to all code found in this distribution, be it the RC
  lhash, DES, etc., code; not just the SSL code. The SSL do
  included with this distribution is covered by the same cop
  except that the holder is Tim Hudson (tjh@cryptsoft.com).

* Copyright remains Eric Young's, and as such any Copyright
  the code are not to be removed.
* If this package is used in a product, Eric Young should be
  as the author of the parts of the library used.
* This can be in the form of a textual message at program st
  in documentation (online or textual) provided with the pac

* Redistribution and use in source and binary forms, with or
  modification, are permitted provided that the following co
  are met:
* 1. Redistributions of source code must retain the copyright
    notice, this list of conditions and the following discl
* 2. Redistributions in binary form must reproduce the above
    notice, this list of conditions and the following discl
    documentation and/or other materials provided with the
* 3. All advertising materials mentioning features or use of
    must display the following acknowledgement:
    "This product includes cryptographic software written by
    Eric Young (eay@cryptsoft.com)"
* The word 'cryptographic' can be left out if the rouines
* being used are not cryptographic related :-)).
* 4. If you include any Windows specific code (or a derivat
    the apps directory (application code) you must include
expat

The **pyexpat** extension is built using an included copy of the expat sources unless the build is configured **--with-system-expat**: 

---

Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY DAMAGE.
libffi

The _ctypes extension is built using an included copy of the libffi sources unless the build is configured --with-system-libffi:

Copyright (c) 1996-2008 Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obt a copy of this software and associated documentation files (the `Software'), to deal in the Software without restriction, inc without limitation the rights to use, copy, modify, merge, publ distribute, sublicense, and/or sell copies of the Software, and permit persons to whom the Software is furnished to do so, subj the following conditions:

The above copyright notice and this permission notice shall be in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIN EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FR OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

zlib

The zlib extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

Copyright (C) 1995-2010 Jean-loup Gailly and Mark Adler
This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this in a product, an acknowledgment in the product documentation would be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and not be misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly          Mark Adler
jloup@gzip.org            madler@alumni.caltech.edu
Index

Index pages by letter:

Symbols | _ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Full index on one page (can be huge)
What’s New in Python

The “What’s New in Python” series of essays takes tours through the most important changes between major Python versions. They are a “must read” for anyone wishing to stay up-to-date after a new release.

- **What’s New in Python 2.7**
  - The Future for Python 2.x
  - Changes to the Handling of Deprecation Warnings
  - Python 3.1 Features
  - PEP 372: Adding an Ordered Dictionary to collections
  - PEP 378: Format Specifier for Thousands Separator
  - PEP 389: The argparse Module for Parsing Command Lines
  - PEP 391: Dictionary-Based Configuration For Logging
  - PEP 3106: Dictionary Views
  - PEP 3137: The memoryview Object
  - Other Language Changes
  - New and Improved Modules
  - Build and C API Changes
  - Other Changes and Fixes
  - Porting to Python 2.7
  - New Features Added to Python 2.7 Maintenance Releases
  - Acknowledgements

- **What’s New in Python 2.6**
  - Python 3.0
  - Changes to the Development Process
  - PEP 343: The ‘with’ statement
  - PEP 366: Explicit Relative Imports From a Main Module
  - PEP 370: Per-user site-packages Directory
  - PEP 371: The multiprocessing Package
  - PEP 3101: Advanced String Formatting
- PEP 3105: `print` As a Function
- PEP 3110: Exception-Handling Changes
- PEP 3112: Byte Literals
- PEP 3116: New I/O Library
- PEP 3118: Revised Buffer Protocol
- PEP 3119: Abstract Base Classes
- PEP 3127: Integer Literal Support and Syntax
- PEP 3129: Class Decorators
- PEP 3141: A Type Hierarchy for Numbers
- Other Language Changes
- New and Improved Modules
- Deprecations and Removals
- Build and C API Changes
- Porting to Python 2.6
- Acknowledgements

- **What’s New in Python 2.5**
  - PEP 308: Conditional Expressions
  - PEP 309: Partial Function Application
  - PEP 314: Metadata for Python Software Packages v1.1
  - PEP 328: Absolute and Relative Imports
  - PEP 338: Executing Modules as Scripts
  - PEP 341: Unified `try/except/finally`
  - PEP 342: New Generator Features
  - PEP 343: The ‘with’ statement
  - PEP 352: Exceptions as New-Style Classes
  - PEP 353: Using `ssize_t` as the index type
  - PEP 357: The ‘__index__’ method
- Other Language Changes
- New, Improved, and Removed Modules
- Build and C API Changes
- Porting to Python 2.5
- Acknowledgements

- **What’s New in Python 2.4**
  - PEP 218: Built-In Set Objects
- PEP 237: Unifying Long Integers and Integers
- PEP 289: Generator Expressions
- PEP 292: Simpler String Substitutions
- PEP 318: Decorators for Functions and Methods
- PEP 322: Reverse Iteration
- PEP 324: New subprocess Module
- PEP 327: Decimal Data Type
- PEP 328: Multi-line Imports
- PEP 331: Locale-Independent Float/String Conversions
- Other Language Changes
  - New, Improved, and Deprecated Modules
  - Build and C API Changes
  - Porting to Python 2.4
  - Acknowledgements
- What’s New in Python 2.3
  - PEP 218: A Standard Set Datatype
  - PEP 255: Simple Generators
  - PEP 263: Source Code Encodings
  - PEP 273: Importing Modules from ZIP Archives
  - PEP 277: Unicode file name support for Windows NT
  - PEP 278: Universal Newline Support
  - PEP 279: enumerate()
  - PEP 282: The logging Package
  - PEP 285: A Boolean Type
  - PEP 293: Codec Error Handling Callbacks
  - PEP 301: Package Index and Metadata for Distutils
  - PEP 302: New Import Hooks
  - PEP 305: Comma-separated Files
  - PEP 307: Pickle Enhancements
  - Extended Slices
  - Other Language Changes
  - New, Improved, and Deprecated Modules
  - Pymalloc: A Specialized Object Allocator
  - Build and C API Changes
- Other Changes and Fixes
- Porting to Python 2.3
- Acknowledgements

- What’s New in Python 2.2
  - Introduction
  - PEPs 252 and 253: Type and Class Changes
  - PEP 234: Iterators
  - PEP 255: Simple Generators
  - PEP 237: Unifying Long Integers and Integers
  - PEP 238: Changing the Division Operator
  - Unicode Changes
  - PEP 227: Nested Scopes
  - New and Improved Modules
  - Interpreter Changes and Fixes
  - Other Changes and Fixes
  - Acknowledgements

- What’s New in Python 2.1
  - Introduction
  - PEP 227: Nested Scopes
  - PEP 236: __future__ Directives
  - PEP 207: Rich Comparisons
  - PEP 230: Warning Framework
  - PEP 229: New Build System
  - PEP 205: Weak References
  - PEP 232: Function Attributes
  - PEP 235: Importing Modules on Case-Insensitive Platforms
  - PEP 217: Interactive Display Hook
  - PEP 208: New Coercion Model
  - PEP 241: Metadata in Python Packages
  - New and Improved Modules
  - Other Changes and Fixes
  - Acknowledgements

- What’s New in Python 2.0
  - Introduction
- What About Python 1.6?
- New Development Process
- Unicode
- List Comprehensions
- Augmented Assignment
- String Methods
- Garbage Collection of Cycles
- Other Core Changes
- Porting to 2.0
- Extending/Embedding Changes
- Distutils: Making Modules Easy to Install
- XML Modules
- Module changes
- New modules
- IDLE Improvements
- Deleted and Deprecated Modules
- Acknowledgements
The Python Tutorial

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python’s elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, https://www.python.org/, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications.

This tutorial introduces the reader informally to the basic concepts and features of the Python language and system. It helps to have a Python interpreter handy for hands-on experience, but all examples are self-contained, so the tutorial can be read off-line as well.

For a description of standard objects and modules, see The Python Standard Library. The Python Language Reference gives a more formal definition of the language. To write extensions in C or C++, read Extending and Embedding the Python Interpreter and Python/C API Reference Manual. There are also several books covering Python in depth.

This tutorial does not attempt to be comprehensive and cover every
single feature, or even every commonly used feature. Instead, it introduces many of Python's most noteworthy features, and will give you a good idea of the language's flavor and style. After reading it, you will be able to read and write Python modules and programs, and you will be ready to learn more about the various Python library modules described in The Python Standard Library.

The Glossary is also worth going through.

- 1. Whetting Your Appetite
- 2. Using the Python Interpreter
  - 2.1. Invoking the Interpreter
    - 2.1.1. Argument Passing
    - 2.1.2. Interactive Mode
  - 2.2. The Interpreter and Its Environment
    - 2.2.1. Source Code Encoding
- 3. An Informal Introduction to Python
  - 3.1. Using Python as a Calculator
    - 3.1.1. Numbers
    - 3.1.2. Strings
    - 3.1.3. Unicode Strings
    - 3.1.4. Lists
  - 3.2. First Steps Towards Programming
- 4. More Control Flow Tools
  - 4.1. if Statements
  - 4.2. for Statements
  - 4.3. The range() Function
  - 4.4. break and continue Statements, and else Clauses on Loops
  - 4.5. pass Statements
  - 4.6. Defining Functions
  - 4.7. More on Defining Functions
    - 4.7.1. Default Argument Values
    - 4.7.2. Keyword Arguments
• 4.7.3. Arbitrary Argument Lists
• 4.7.4. Unpacking Argument Lists
• 4.7.5. Lambda Expressions
• 4.7.6. Documentation Strings
○ 4.8. Intermezzo: Coding Style

• 5. Data Structures
○ 5.1. More on Lists
  ■ 5.1.1. Using Lists as Stacks
  ■ 5.1.2. Using Lists as Queues
  ■ 5.1.3. Functional Programming Tools
  ■ 5.1.4. List Comprehensions
    ■ 5.1.4.1. Nested List Comprehensions
○ 5.2. The `del` statement
○ 5.3. Tuples and Sequences
○ 5.4. Sets
○ 5.5. Dictionaries
○ 5.6. Looping Techniques
○ 5.7. More on Conditions
○ 5.8. Comparing Sequences and Other Types

• 6. Modules
○ 6.1. More on Modules
  ■ 6.1.1. Executing modules as scripts
  ■ 6.1.2. The Module Search Path
  ■ 6.1.3. “Compiled” Python files
○ 6.2. Standard Modules
○ 6.3. The `dir()` Function
○ 6.4. Packages
  ■ 6.4.1. Importing * From a Package
  ■ 6.4.2. Intra-package References
  ■ 6.4.3. Packages in Multiple Directories

• 7. Input and Output
○ 7.1. Fancier Output Formatting
  ■ 7.1.1. Old string formatting
○ 7.2. Reading and Writing Files
7.2.1. Methods of File Objects
7.2.2. Saving structured data with json

- 8. Errors and Exceptions
  - 8.1. Syntax Errors
  - 8.2. Exceptions
  - 8.3. Handling Exceptions
  - 8.4. Raising Exceptions
  - 8.5. User-defined Exceptions
  - 8.6. Defining Clean-up Actions
  - 8.7. Predefined Clean-up Actions

- 9. Classes
  - 9.1. A Word About Names and Objects
  - 9.2. Python Scopes and Namespaces
  - 9.3. A First Look at Classes
    - 9.3.1. Class Definition Syntax
    - 9.3.2. Class Objects
    - 9.3.3. Instance Objects
    - 9.3.4. Method Objects
    - 9.3.5. Class and Instance Variables
  - 9.4. Random Remarks
  - 9.5. Inheritance
    - 9.5.1. Multiple Inheritance
  - 9.6. Private Variables and Class-local References
  - 9.7. Odds and Ends
  - 9.8. Exceptions Are Classes Too
  - 9.9. Iterators
  - 9.10. Generators
  - 9.11. Generator Expressions

- 10. Brief Tour of the Standard Library
  - 10.1. Operating System Interface
  - 10.2. File Wildcards
  - 10.3. Command Line Arguments
  - 10.4. Error Output Redirection and Program Termination
  - 10.5. String Pattern Matching
• 10.6. Mathematics
• 10.7. Internet Access
• 10.8. Dates and Times
• 10.9. Data Compression
• 10.10. Performance Measurement
• 10.11. Quality Control
• 10.12. Batteries Included
• 11. Brief Tour of the Standard Library – Part II
  • 11.1. Output Formatting
  • 11.2. Templating
  • 11.3. Working with Binary Data Record Layouts
  • 11.4. Multi-threading
  • 11.5. Logging
  • 11.6. Weak References
  • 11.7. Tools for Working with Lists
  • 11.8. Decimal Floating Point Arithmetic
• 12. What Now?
• 13. Interactive Input Editing and History Substitution
  • 13.1. Line Editing
  • 13.2. History Substitution
  • 13.3. Key Bindings
  • 13.4. Alternatives to the Interactive Interpreter
• 14. Floating Point Arithmetic: Issues and Limitations
  • 14.1. Representation Error
• 15. Appendix
  • 15.1. Interactive Mode
    ■ 15.1.1. Error Handling
    ■ 15.1.2. Executable Python Scripts
    ■ 15.1.3. The Interactive Startup File
    ■ 15.1.4. The Customization Modules
While The Python Language Reference describes the exact syntax and semantics of the Python language, this library reference manual describes the standard library that is distributed with Python. It also describes some of the optional components that are commonly included in Python distributions.

Python’s standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed below. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.

The Python installers for the Windows platform usually include the entire standard library and often also include many additional components. For Unix-like operating systems Python is normally provided as a collection of packages, so it may be necessary to use the packaging tools provided with the operating system to obtain some or all of the optional components.

In addition to the standard library, there is a growing collection of several thousand components (from individual programs and modules to packages and entire application development frameworks), available from the Python Package Index.

- 1. Introduction
- 2. Built-in Functions
- 3. Non-essential Built-in Functions
• 4. Built-in Constants
  ○ 4.1. Constants added by the site module

• 5. Built-in Types
  ○ 5.1. Truth Value Testing
  ○ 5.2. Boolean Operations — and, or, not
  ○ 5.3. Comparisons
  ○ 5.4. Numeric Types — int, float, long, complex
  ○ 5.5. Iterator Types
  ○ 5.6. Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange
  ○ 5.7. Set Types — set, frozenset
  ○ 5.8. Mapping Types — dict
  ○ 5.9. File Objects
  ○ 5.10. memoryview type
  ○ 5.11. Context Manager Types
  ○ 5.12. Other Built-in Types
  ○ 5.13. Special Attributes

• 6. Built-in Exceptions
  ○ 6.1. Exception hierarchy

• 7. String Services
  ○ 7.1. string — Common string operations
  ○ 7.2. re — Regular expression operations
  ○ 7.3. struct — Interpret strings as packed binary data
  ○ 7.4. difflib — Helpers for computing deltas
  ○ 7.5. StringIO — Read and write strings as files
  ○ 7.6. cStringIO — Faster version of StringIO
  ○ 7.7. textwrap — Text wrapping and filling
  ○ 7.8. codecs — Codec registry and base classes
  ○ 7.9. unicodedata — Unicode Database
  ○ 7.10. stringprep — Internet String Preparation
  ○ 7.11. fpformat — Floating point conversions

• 8. Data Types
8.1. **datetime** — Basic date and time types
8.2. **calendar** — General calendar-related functions
8.3. **collections** — High-performance container datatypes
8.4. **heapq** — Heap queue algorithm
8.5. **bisect** — Array bisection algorithm
8.6. **array** — Efficient arrays of numeric values
8.7. **sets** — Unordered collections of unique elements
8.8. **sched** — Event scheduler
8.9. **mutex** — Mutual exclusion support
8.10. **Queue** — A synchronized queue class
8.11. **weakref** — Weak references
8.12. **UserDict** — Class wrapper for dictionary objects
8.13. **UserList** — Class wrapper for list objects
8.14. **UserString** — Class wrapper for string objects
8.15. **types** — Names for built-in types
8.16. **new** — Creation of runtime internal objects
8.17. **copy** — Shallow and deep copy operations
8.18. **pprint** — Data pretty printer
8.19. **repr** — Alternate `repr()` implementation

9. Numeric and Mathematical Modules
9.1. **numbers** — Numeric abstract base classes
9.2. **math** — Mathematical functions
9.3. **cmath** — Mathematical functions for complex numbers
9.4. **decimal** — Decimal fixed point and floating point arithmetic
9.5. **fractions** — Rational numbers
9.6. **random** — Generate pseudo-random numbers
9.7. **itertools** — Functions creating iterators for efficient looping
9.8. **functools** — Higher-order functions and operations on callable objects
9.9. **operator** — Standard operators as functions

- **10. File and Directory Access**
  - 10.1. **os.path** — Common pathname manipulations
  - 10.2. **fileinput** — Iterate over lines from multiple input streams
  - 10.3. **stat** — Interpreting `stat()` results
  - 10.4. **statvfs** — Constants used with `os.statvfs()`
  - 10.5. **filecmp** — File and Directory Comparisons
  - 10.6. **tempfile** — Generate temporary files and directories
  - 10.7. **glob** — Unix style pathname pattern expansion
  - 10.8. **fnmatch** — Unix filename pattern matching
  - 10.9. **linecache** — Random access to text lines
  - 10.10. **shutil** — High-level file operations
  - 10.11. **dircache** — Cached directory listings
  - 10.12. **macpath** — Mac OS 9 path manipulation functions

- **11. Data Persistence**
  - 11.1. **pickle** — Python object serialization
  - 11.2. **cPickle** — A faster `pickle`
  - 11.3. **copy_reg** — Register `pickle` support functions
  - 11.4. **shelve** — Python object persistence
  - 11.5. **marshal** — Internal Python object serialization
  - 11.6. **anydbm** — Generic access to DBM-style databases
  - 11.7. **whichdb** — Guess which DBM module created a database
  - 11.8. **dbm** — Simple “database” interface
  - 11.9. **gdbm** — GNU’s reinterpretation of dbm
  - 11.10. **dbhash** — DBM-style interface to the BSD database library
  - 11.11. **bsddb** — Interface to Berkeley DB library
  - 11.12. **dumbdbm** — Portable DBM implementation
  - 11.13. **sqlite3** — DB-API 2.0 interface for SQLite
databases

- **12. Data Compression and Archiving**
  - 12.1. **zlib** — Compression compatible with **gzip**
  - 12.2. **gzip** — Support for **gzip** files
  - 12.3. **bz2** — Compression compatible with **bzip2**
  - 12.4. **zipfile** — Work with ZIP archives
  - 12.5. **tarfile** — Read and write tar archive files

- **13. File Formats**
  - 13.1. **csv** — CSV File Reading and Writing
  - 13.2. **ConfigParser** — Configuration file parser
  - 13.3. **robotparser** — Parser for robots.txt
  - 13.4. **netrc** — netrc file processing
  - 13.5. **xdrplib** — Encode and decode XDR data
  - 13.6. **plistlib** — Generate and parse Mac OS X .plist files

- **14. Cryptographic Services**
  - 14.1. **hashlib** — Secure hashes and message digests
  - 14.2. **hmac** — Keyed-Hashing for Message Authentication
  - 14.3. **md5** — MD5 message digest algorithm
  - 14.4. **sha** — SHA-1 message digest algorithm

- **15. Generic Operating System Services**
  - 15.1. **os** — Miscellaneous operating system interfaces
  - 15.2. **io** — Core tools for working with streams
  - 15.3. **time** — Time access and conversions
  - 15.4. **argparse** — Parser for command-line options, arguments and sub-commands
  - 15.5. **optparse** — Parser for command line options
  - 15.6. **getopt** — C-style parser for command line options
  - 15.7. **logging** — Logging facility for Python
  - 15.8. **logging.config** — Logging configuration
  - 15.9. **logging.handlers** — Logging handlers
• 15.10. `getpass` — Portable password input
• 15.11. `curses` — Terminal handling for character-cell displays
• 15.12. `curses.textpad` — Text input widget for curses programs
• 15.13. `curses.ascii` — Utilities for ASCII characters
• 15.14. `curses.panel` — A panel stack extension for curses
• 15.15. `platform` — Access to underlying platform’s identifying data
• 15.16. `errno` — Standard errno system symbols
• 15.17. `ctypes` — A foreign function library for Python

• 16. Optional Operating System Services
  • 16.1. `select` — Waiting for I/O completion
  • 16.2. `threading` — Higher-level threading interface
  • 16.3. `thread` — Multiple threads of control
  • 16.4. `dummy_threading` — Drop-in replacement for the `threading` module
  • 16.5. `dummy_thread` — Drop-in replacement for the `thread` module
  • 16.6. `multiprocessing` — Process-based “threading” interface
  • 16.7. `mmap` — Memory-mapped file support
  • 16.8. `readline` — GNU readline interface
  • 16.9. `rlcompleter` — Completion function for GNU readline

• 17. Interprocess Communication and Networking
  • 17.1. `subprocess` — Subprocess management
  • 17.2. `socket` — Low-level networking interface
  • 17.3. `ssl` — TLS/SSL wrapper for socket objects
  • 17.4. `signal` — Set handlers for asynchronous events
  • 17.5. `popen2` — Subprocesses with accessible I/O streams
  • 17.6. `asyncore` — Asynchronous socket handler
17.7. `asynchat` — Asynchronous socket command/response handler

18. Internet Data Handling
- 18.1. `email` — An email and MIME handling package
- 18.2. `json` — JSON encoder and decoder
- 18.3. `mailcap` — Mailcap file handling
- 18.4. `mailbox` — Manipulate mailboxes in various formats
- 18.5. `mhlib` — Access to MH mailboxes
- 18.6. `mimetools` — Tools for parsing MIME messages
- 18.7. `mimetypes` — Map filenames to MIME types
- 18.8. `MimeWriter` — Generic MIME file writer
- 18.9. `mimify` — MIME processing of mail messages
- 18.10. `multifile` — Support for files containing distinct parts
- 18.11. `rfc822` — Parse RFC 2822 mail headers
- 18.12. `base64` — RFC 3548: Base16, Base32, Base64 Data Encodings
- 18.13. `binhex` — Encode and decode binhex4 files
- 18.14. `binascii` — Convert between binary and ASCII
- 18.15. `quopri` — Encode and decode MIME quoted-printable data
- 18.16. `uu` — Encode and decode uuencode files

19. Structured Markup Processing Tools
- 19.1. `HTMLParser` — Simple HTML and XHTML parser
- 19.2. `sgmllib` — Simple SGML parser
- 19.3. `httplib` — A parser for HTML documents
- 19.4. `htmlentitydefs` — Definitions of HTML general entities
- 19.5. XML Processing Modules
- 19.6. XML vulnerabilities
- 19.7. `xml.etree.ElementTree` — The ElementTree XML API
19.8. `xml.dom` — The Document Object Model API
19.9. `xml.dom.minidom` — Minimal DOM implementation
19.10. `xml.dom.pulldom` — Support for building partial DOM trees
19.11. `xml.sax` — Support for SAX2 parsers
19.12. `xml.sax.handler` — Base classes for SAX handlers
19.13. `xml.sax.saxutils` — SAX Utilities
19.14. `xml.sax.xmlreader` — Interface for XML parsers
19.15. `xml.parsers.expat` — Fast XML parsing using Expat

20. Internet Protocols and Support
20.1. `webbrowser` — Convenient Web-browser controller
20.2. `cgi` — Common Gateway Interface support
20.3. `cgitb` — Traceback manager for CGI scripts
20.4. `wsgiref` — WSGI Utilities and Reference Implementation
20.5. `urllib` — Open arbitrary resources by URL
20.6. `urllib2` — extensible library for opening URLs
20.7. `httplib` — HTTP protocol client
20.8. `ftplib` — FTP protocol client
20.9. `poplib` — POP3 protocol client
20.10. `imaplib` — IMAP4 protocol client
20.11. `nntplib` — NNTP protocol client
20.12. `smtplib` — SMTP protocol client
20.13. `smtpd` — SMTP Server
20.14. `telnetlib` — Telnet client
20.15. `uuid` — UUID objects according to RFC 4122
20.16. `urlparse` — Parse URLs into components
20.17. `SocketServer` — A framework for network servers
20.18. `BaseHTTPServer` — Basic HTTP server
20.19. `SimpleHTTPServer` — Simple HTTP request handler
20.20. `CGIHTTPServer` — CGI-capable HTTP request
• 20.21. cookie — Cookie handling for HTTP clients
• 20.22. Cookie — HTTP state management
• 20.23. xmlrpclib — XML-RPC client access
• 20.24. SimpleXMLRPCServer — Basic XML-RPC server
• 20.25. DocXMLRPCServer — Self-documenting XML-RPC server

• 21. Multimedia Services
  • 21.1. audioop — Manipulate raw audio data
  • 21.2. imageop — Manipulate raw image data
  • 21.3. aifc — Read and write AIFF and AIFC files
  • 21.4. sunau — Read and write Sun AU files
  • 21.5. wave — Read and write WAV files
  • 21.6. chunk — Read IFF chunked data
  • 21.7. colorsys — Conversions between color systems
  • 21.8. imghdr — Determine the type of an image
  • 21.9. sndhdr — Determine type of sound file
  • 21.10. ossaudiodev — Access to OSS-compatible audio devices

• 22. Internationalization
  • 22.1. gettext — Multilingual internationalization services
  • 22.2. locale — Internationalization services

• 23. Program Frameworks
  • 23.1. cmd — Support for line-oriented command interpreters
  • 23.2. shlex — Simple lexical analysis

• 24. Graphical User Interfaces with Tk
  • 24.1. Tkinter — Python interface to Tcl/Tk
  • 24.2. ttk — Tk themed widgets
  • 24.3. Tix — Extension widgets for Tk
  • 24.4. ScrolledText — Scrolled Text Widget
  • 24.5. turtle — Turtle graphics for Tk
24.6. IDLE
24.7. Other Graphical User Interface Packages

25. Development Tools
25.1. pydoc — Documentation generator and online help system
25.2. doctest — Test interactive Python examples
25.3. unittest — Unit testing framework
25.4. 2to3 - Automated Python 2 to 3 code translation
25.5. test — Regression tests package for Python
25.6. test.test_support — Utility functions for tests

26. Debugging and Profiling
26.1. bdb — Debugger framework
26.2. pdb — The Python Debugger
26.3. Debugger Commands
26.4. The Python Profilers
26.5. hotshot — High performance logging profiler
26.6. timeit — Measure execution time of small code snippets
26.7. trace — Trace or track Python statement execution

27. Software Packaging and Distribution
27.1. distutils — Building and installing Python modules
27.2. ensurepip — Bootstrapping the pip installer

28. Python Runtime Services
28.1. sys — System-specific parameters and functions
28.2. sysconfig — Provide access to Python’s configuration information
28.3. __builtin__ — Built-in objects
28.4. future_builtins — Python 3 builtins
28.5. __main__ — Top-level script environment
28.6. warnings — Warning control
28.7. contextlib — Utilities for with-statement contexts
28.8. abc — Abstract Base Classes
28.9. `atexit` — Exit handlers
28.10. `traceback` — Print or retrieve a stack traceback
28.11. `__future__` — Future statement definitions
28.12. `gc` — Garbage Collector interface
28.13. `inspect` — Inspect live objects
28.14. `site` — Site-specific configuration hook
28.15. `user` — User-specific configuration hook
28.16. `fpectl` — Floating point exception control

29. Custom Python Interpreters
29.1. `code` — Interpreter base classes
29.2. `codeop` — Compile Python code

30. Restricted Execution
30.1. `rexec` — Restricted execution framework
30.2. `Bastion` — Restricting access to objects

31. Importing Modules
31.1. `imp` — Access the `import` internals
31.2. `importlib` — Convenience wrappers for `__import__()`
31.3. `imputil` — Import utilities
31.4. `zipimport` — Import modules from Zip archives
31.5. `pkgutil` — Package extension utility
31.6. `modulefinder` — Find modules used by a script
31.7. `runpy` — Locating and executing Python modules

32. Python Language Services
32.1. `parser` — Access Python parse trees
32.2. `ast` — Abstract Syntax Trees
32.3. `symtable` — Access to the compiler’s symbol tables
32.4. `symbol` — Constants used with Python parse trees
32.5. `token` — Constants used with Python parse trees
32.6. `keyword` — Testing for Python keywords
32.7. `tokenize` — Tokenizer for Python source
32.8. `tabnanny` — Detection of ambiguous indentation
32.9. pyc1br — Python class browser support
32.10. py_compile — Compile Python source files
32.11. compileall — Byte-compile Python libraries
32.12. dis — Disassembler for Python bytecode
32.13. pickletools — Tools for pickle developers

33. Python compiler package
   33.1. The basic interface
   33.2. Limitations
   33.3. Python Abstract Syntax
   33.4. Using Visitors to Walk ASTs
   33.5. Bytecode Generation

34. Miscellaneous Services
   34.1. formatter — Generic output formatting

35. MS Windows Specific Services
   35.1. msilib — Read and write Microsoft Installer files
   35.2. msvcrt — Useful routines from the MS VC++ runtime
   35.3. _winreg — Windows registry access
   35.4. winsound — Sound-playing interface for Windows

36. Unix Specific Services
   36.1. posix — The most common POSIX system calls
   36.2. pwd — The password database
   36.3. spwd — The shadow password database
   36.4. grp — The group database
   36.5. crypt — Function to check Unix passwords
   36.6. dl — Call C functions in shared objects
   36.7. termios — POSIX style tty control
   36.8. tty — Terminal control functions
   36.9. pty — Pseudo-terminal utilities
   36.10. fcntl — The fcntl and ioctl system calls
   36.11. pipes — Interface to shell pipelines
   36.12. posixfile — File-like objects with locking support
- 36.13. resource — Resource usage information
- 36.14. nis — Interface to Sun's NIS (Yellow Pages)
- 36.15. syslog — Unix syslog library routines
- 36.16. commands — Utilities for running commands

- 37. Mac OS X specific services
  - 37.1. ic — Access to the Mac OS X Internet Config
  - 37.2. MacOS — Access to Mac OS interpreter features
  - 37.3. macostools — Convenience routines for file manipulation
  - 37.4. findertools — The finder's Apple Events interface
  - 37.5. EasyDialogs — Basic Macintosh dialogs
  - 37.6. FrameWork — Interactive application framework
  - 37.7. autoGIL — Global Interpreter Lock handling in event loops
  - 37.8. Mac OS Toolbox Modules
  - 37.9. ColorPicker — Color selection dialog

- 38. MacPython OSA Modules
  - 38.1. gensuitemodule — Generate OSA stub packages
  - 38.2. aetools — OSA client support
  - 38.3. aepack — Conversion between Python variables and AppleEvent data containers
  - 38.4. aetypes — AppleEvent objects
  - 38.5. MiniAEFrame — Open Scripting Architecture server support

- 39. SGI IRIX Specific Services
  - 39.1. a1 — Audio functions on the SGI
  - 39.2. AL — Constants used with the a1 module
  - 39.3. cd — CD-ROM access on SGI systems
  - 39.4. f1 — FORMS library for graphical user interfaces
  - 39.5. FL — Constants used with the f1 module
  - 39.6. flp — Functions for loading stored FORMS designs
39.7. fm — *Font Manager* interface
39.8. gl — *Graphics Library* interface
39.9. **DEVICE** — Constants used with the **gl** module
39.10. **GL** — Constants used with the **gl** module
39.11. **imgfile** — Support for SGI imglib files
39.12. **jpeg** — Read and write JPEG files

- **40. SunOS Specific Services**
  - 40.1. **sunaudiodev** — Access to Sun audio hardware
  - 40.2. **SUNAUDIODEV** — Constants used with **sunaudiodev**

- **41. Undocumented Modules**
  - 41.1. Miscellaneous useful utilities
  - 41.2. Platform specific modules
  - 41.3. Multimedia
  - 41.4. Undocumented Mac OS modules
  - 41.5. Obsolete
  - 41.6. SGI-specific Extension modules
The Python Language Reference

This reference manual describes the syntax and “core semantics” of the language. It is terse, but attempts to be exact and complete. The semantics of non-essential built-in object types and of the built-in functions and modules are described in The Python Standard Library. For an informal introduction to the language, see The Python Tutorial. For C or C++ programmers, two additional manuals exist: Extending and Embedding the Python Interpreter describes the high-level picture of how to write a Python extension module, and the Python/C API Reference Manual describes the interfaces available to C/C++ programmers in detail.

- 1. Introduction
  - 1.1. Alternate Implementations
  - 1.2. Notation
- 2. Lexical analysis
  - 2.1. Line structure
  - 2.2. Other tokens
  - 2.3. Identifiers and keywords
  - 2.4. Literals
  - 2.5. Operators
  - 2.6. Delimiters
- 3. Data model
  - 3.1. Objects, values and types
  - 3.2. The standard type hierarchy
  - 3.3. New-style and classic classes
  - 3.4. Special method names
- 4. Execution model
  - 4.1. Naming and binding
  - 4.2. Exceptions
- 5. Expressions
  - 5.1. Arithmetic conversions
• 5.2. Atoms
• 5.3. Primaries
• 5.4. The power operator
• 5.5. Unary arithmetic and bitwise operations
• 5.6. Binary arithmetic operations
• 5.7. Shifting operations
• 5.8. Binary bitwise operations
• 5.9. Comparisons
• 5.10. Boolean operations
• 5.11. Conditional Expressions
• 5.12. Lambdas
• 5.13. Expression lists
• 5.14. Evaluation order
• 5.15. Operator precedence

• 6. Simple statements
  • 6.1. Expression statements
  • 6.2. Assignment statements
  • 6.3. The assert statement
  • 6.4. The pass statement
  • 6.5. The del statement
  • 6.6. The print statement
  • 6.7. The return statement
  • 6.8. The yield statement
  • 6.9. The raise statement
  • 6.10. The break statement
  • 6.11. The continue statement
  • 6.12. The import statement
  • 6.13. The global statement
  • 6.14. The exec statement

• 7. Compound statements
  • 7.1. The if statement
  • 7.2. The while statement
- 7.3. The `for` statement
- 7.4. The `try` statement
- 7.5. The `with` statement
- 7.6. Function definitions
- 7.7. Class definitions
- 8. Top-level components
  - 8.1. Complete Python programs
  - 8.2. File input
  - 8.3. Interactive input
  - 8.4. Expression input
- 9. Full Grammar specification
Python Setup and Usage

This part of the documentation is devoted to general information on the setup of the Python environment on different platforms, the invocation of the interpreter and things that make working with Python easier.

- 1. Command line and environment
  - 1.1. Command line
    - 1.1.1. Interface options
    - 1.1.2. Generic options
    - 1.1.3. Miscellaneous options
    - 1.1.4. Options you shouldn’t use
  - 1.2. Environment variables
    - 1.2.1. Debug-mode variables
- 2. Using Python on Unix platforms
  - 2.1. Getting and installing the latest version of Python
    - 2.1.1. On Linux
    - 2.1.2. On FreeBSD and OpenBSD
    - 2.1.3. On OpenSolaris
  - 2.2. Building Python
  - 2.3. Python-related paths and files
  - 2.4. Miscellaneous
  - 2.5. Editors
- 3. Using Python on Windows
  - 3.1. Installing Python
  - 3.2. Alternative bundles
  - 3.3. Configuring Python
    - 3.3.1. Excursus: Setting environment variables
    - 3.3.2. Finding the Python executable
    - 3.3.3. Finding modules
    - 3.3.4. Executing scripts
  - 3.4. Additional modules
3.4.1. PyWin32
3.4.2. Py2exe
3.4.3. WConio
3.5. Compiling Python on Windows
3.6. Other resources

4. Using Python on a Macintosh
4.1. Getting and Installing MacPython
   4.1.1. How to run a Python script
   4.1.2. Running scripts with a GUI
   4.1.3. Configuration
4.2. The IDE
4.3. Installing Additional Python Packages
4.4. GUI Programming on the Mac
4.5. Distributing Python Applications on the Mac
4.6. Other Resources
Python HOWTOs

Python HOWTOs are documents that cover a single, specific topic, and attempt to cover it fairly completely. Modelled on the Linux Documentation Project’s HOWTO collection, this collection is an effort to foster documentation that’s more detailed than the Python Library Reference.

Currently, the HOWTOs are:

- Porting Python 2 Code to Python 3
- Porting Extension Modules to Python 3
- Curses Programming with Python
- Descriptor HowTo Guide
- Idioms and Anti-Idioms in Python
- Functional Programming HOWTO
- Logging HOWTO
- Logging Cookbook
- Regular Expression HOWTO
- Socket Programming HOWTO
- Sorting HOW TO
- Unicode HOWTO
- HOWTO Fetch Internet Resources Using urllib2
- HOWTO Use Python in the web
- Argparse Tutorial
Installing Python Modules

Email: distutils-sig@python.org

As a popular open source development project, Python has an active supporting community of contributors and users that also make their software available for other Python developers to use under open source license terms.

This allows Python users to share and collaborate effectively, benefiting from the solutions others have already created to common (and sometimes even rare!) problems, as well as potentially contributing their own solutions to the common pool.

This guide covers the installation part of the process. For a guide to creating and sharing your own Python projects, refer to the distribution guide.

Note: For corporate and other institutional users, be aware that many organisations have their own policies around using and contributing to open source software. Please take such policies into account when making use of the distribution and installation tools provided with Python.
Key terms

- **pip** is the preferred installer program. Starting with Python 2.7.9, it is included by default with the Python binary installers.
- A virtual environment is a semi-isolated Python environment that allows packages to be installed for use by a particular application, rather than being installed system-wide.
- **virtualenv** is a third party tool for creating virtual environments; it defaults to installing `pip` into all created virtual environments.
- The **Python Packaging Index** is a public repository of open source licensed packages made available for use by other Python users.
- The **Python Packaging Authority** are the group of developers and documentation authors responsible for the maintenance and evolution of the standard packaging tools and the associated metadata and file format standards. They maintain a variety of tools, documentation and issue trackers on both GitHub and BitBucket.
- **distutils** is the original build and distribution system first added to the Python standard library in 1998. While direct use of **distutils** is being phased out, it still laid the foundation for the current packaging and distribution infrastructure, and it not only remains part of the standard library, but its name lives on in other ways (such as the name of the mailing list used to coordinate Python packaging standards development).
Basic usage

The standard packaging tools are all designed to be used from the command line.

The following command will install the latest version of a module and its dependencies from the Python Packaging Index:

```
python -m pip install SomePackage
```

**Note:** For POSIX users (including Mac OS X and Linux users), the examples in this guide assume the use of a virtual environment. You may install virtualenv to provide such environments using either pip (``pip install virtualenv``) or through your system package manager (commonly called virtualenv or python-virtualenv).

For Windows users, the examples in this guide assume that the option to adjust the system PATH environment variable was selected when installing Python.

It's also possible to specify an exact or minimum version directly on the command line. When using comparator operators such as `>`, `<` or some other special character which get interpreted by shell, the package name and the version should be enclosed within double quotes:

```
python -m pip install SomePackage==1.0.4    # specific version
python -m pip install "SomePackage>=1.0.4"  # minimum version
```

Normally, if a suitable module is already installed, attempting to install it again will have no effect. Upgrading existing modules must be requested explicitly:
python -m pip install --upgrade SomePackage

More information and resources regarding pip and its capabilities can be found in the Python Packaging User Guide.

See also: Python Packaging User Guide: Installing Python Distribution Packages
How do I ...?

These are quick answers or links for some common tasks.

... install `pip` in versions of Python prior to Python 2.7.9?

Python only started bundling `pip` with Python 2.7.9. For earlier versions, `pip` needs to be “bootstrapped” as described in the Python Packaging User Guide.

**See also:** [Python Packaging User Guide: Requirements for Installing Packages](#)

... install packages just for the current user?

Passing the `--user` option to `python -m pip install` will install a package just for the current user, rather than for all users of the system.

... install scientific Python packages?

A number of scientific Python packages have complex binary dependencies, and aren’t currently easy to install using `pip` directly. At this point in time, it will often be easier for users to install these packages by **other means** rather than attempting to install them with `pip`.

**See also:** [Python Packaging User Guide: Installing Scientific Packages](#)
... work with multiple versions of Python installed in parallel?

On Linux, Mac OS X and other POSIX systems, use the versioned Python commands in combination with the `-m` switch to run the appropriate copy of `pip`:

```
python2   -m pip install SomePackage  # default Python 2
python2.7 -m pip install SomePackage  # specifically Python 2.7
python3   -m pip install SomePackage  # default Python 3
python3.4 -m pip install SomePackage  # specifically Python 3.4
```

(appropriately versioned `pip` commands may also be available)

On Windows, use the `py` Python launcher in combination with the `-m` switch:

```
py -2    -m pip install SomePackage  # default Python 2
py -2.7  -m pip install SomePackage  # specifically Python 2.7
py -3    -m pip install SomePackage  # default Python 3
py -3.4  -m pip install SomePackage  # specifically Python 3.4
```
Common installation issues

Installing into the system Python on Linux

On Linux systems, a Python installation will typically be included as part of the distribution. Installing into this Python installation requires root access to the system, and may interfere with the operation of the system package manager and other components of the system if a component is unexpectedly upgraded using `pip`.

On such systems, it is often better to use a virtual environment or a per-user installation when installing packages with `pip`.

Installing binary extensions

Python has typically relied heavily on source based distribution, with end users being expected to compile extension modules from source as part of the installation process.

With the introduction of support for the binary `wheel` format, and the ability to publish wheels for at least Windows and Mac OS X through the Python Packaging Index, this problem is expected to diminish over time, as users are more regularly able to install pre-built extensions rather than needing to build them themselves.

Some of the solutions for installing scientific software that is not yet available as pre-built `wheel` files may also help with obtaining other binary extensions without needing to build them locally.

See also: Python Packaging User Guide: Binary Extensions
As a popular open source development project, Python has an active supporting community of contributors and users that also make their software available for other Python developers to use under open source license terms.

This allows Python users to share and collaborate effectively, benefiting from the solutions others have already created to common (and sometimes even rare!) problems, as well as potentially contributing their own solutions to the common pool.

This guide covers the distribution part of the process. For a guide to installing other Python projects, refer to the installation guide.

**Note:** For corporate and other institutional users, be aware that many organisations have their own policies around using and contributing to open source software. Please take such policies into account when making use of the distribution and installation tools provided with Python.
Key terms

- **the Python Packaging Index** is a public repository of open source licensed packages made available for use by other Python users.
- **the Python Packaging Authority** are the group of developers and documentation authors responsible for the maintenance and evolution of the standard packaging tools and the associated metadata and file format standards. They maintain a variety of tools, documentation and issue trackers on both GitHub and BitBucket.
- **distutils** is the original build and distribution system first added to the Python standard library in 1998. While direct use of **distutils** is being phased out, it still laid the foundation for the current packaging and distribution infrastructure, and it not only remains part of the standard library, but its name lives on in other ways (such as the name of the mailing list used to coordinate Python packaging standards development).
- **setuptools** is a (largely) drop-in replacement for **distutils** first published in 2004. Its most notable addition over the unmodified **distutils** tools was the ability to declare dependencies on other packages. It is currently recommended as a more regularly updated alternative to **distutils** that offers consistent support for more recent packaging standards across a wide range of Python versions.
- **wheel** (in this context) is a project that adds the `bdist_wheel` command to **distutils/setuptools**. This produces a cross platform binary packaging format (called “wheels” or “wheel files” and defined in PEP 427) that allows Python libraries, even those including binary extensions, to be installed on a system without needing to be built locally.
Open source licensing and collaboration

In most parts of the world, software is automatically covered by copyright. This means that other developers require explicit permission to copy, use, modify and redistribute the software.

Open source licensing is a way of explicitly granting such permission in a relatively consistent way, allowing developers to share and collaborate efficiently by making common solutions to various problems freely available. This leaves many developers free to spend more time focusing on the problems that are relatively unique to their specific situation.

The distribution tools provided with Python are designed to make it reasonably straightforward for developers to make their own contributions back to that common pool of software if they choose to do so.

The same distribution tools can also be used to distribute software within an organisation, regardless of whether that software is published as open source software or not.
Installing the tools

The standard library does not include build tools that support modern Python packaging standards, as the core development team has found that it is important to have standard tools that work consistently, even on older versions of Python.

The currently recommended build and distribution tools can be installed by invoking the `pip` module at the command line:

```
python -m pip install setuptools wheel twine
```

**Note:** For POSIX users (including Mac OS X and Linux users), these instructions assume the use of a virtual environment.

For Windows users, these instructions assume that the option to adjust the system PATH environment variable was selected when installing Python.

The Python Packaging User Guide includes more details on the currently recommended tools.
The Python Packaging User Guide covers the various key steps and elements involved in creating a project:

- Project structure
- Building and packaging the project
- Uploading the project to the Python Packaging Index
How do I...?

These are quick answers or links for some common tasks.

... choose a name for my project?

This isn’t an easy topic, but here are a few tips:

- check the Python Packaging Index to see if the name is already in use
- check popular hosting sites like GitHub, BitBucket, etc to see if there is already a project with that name
- check what comes up in a web search for the name you’re considering
- avoid particularly common words, especially ones with multiple meanings, as they can make it difficult for users to find your software when searching for it

... create and distribute binary extensions?

This is actually quite a complex topic, with a variety of alternatives available depending on exactly what you’re aiming to achieve. See the Python Packaging User Guide for more information and recommendations.

**See also:** [Python Packaging User Guide: Binary Extensions](python-packaging-user-guide-binary-extensions)
Extending and Embedding the Python Interpreter

This document describes how to write modules in C or C++ to extend the Python interpreter with new modules. Those modules can not only define new functions but also new object types and their methods. The document also describes how to embed the Python interpreter in another application, for use as an extension language. Finally, it shows how to compile and link extension modules so that they can be loaded dynamically (at run time) into the interpreter, if the underlying operating system supports this feature.

This document assumes basic knowledge about Python. For an informal introduction to the language, see The Python Tutorial. The Python Language Reference gives a more formal definition of the language. The Python Standard Library documents the existing object types, functions and modules (both built-in and written in Python) that give the language its wide application range.

For a detailed description of the whole Python/C API, see the separate Python/C API Reference Manual.

Note: This guide only covers the basic tools for creating extensions provided as part of this version of CPython. Third party tools may offer simpler alternatives. Refer to the binary extensions section in the Python Packaging User Guide for more information.

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

2. Defining New Types
   2.1. The Basics
   2.2. Type Methods

3. Building C and C++ Extensions with distutils
   3.1. Distributing your extension modules

4. Building C and C++ Extensions on Windows
   4.1. A Cookbook Approach
   4.2. Differences Between Unix and Windows
   4.3. Using DLLs in Practice

5. Embedding Python in Another Application
   5.1. Very High Level Embedding
   5.2. Beyond Very High Level Embedding: An overview
   5.3. Pure Embedding
   5.4. Extending Embedded Python
   5.5. Embedding Python in C++
   5.6. Compiling and Linking under Unix-like systems
This manual documents the API used by C and C++ programmers who want to write extension modules or embed Python. It is a companion to Extending and Embedding the Python Interpreter, which describes the general principles of extension writing but does not document the API functions in detail.

- Introduction
  - Include Files
  - Objects, Types and Reference Counts
  - Exceptions
  - Embedding Python
  - Debugging Builds
- The Very High Level Layer
- Reference Counting
- Exception Handling
  - Unicode Exception Objects
  - Recursion Control
  - Standard Exceptions
  - String Exceptions
- Utilities
  - Operating System Utilities
  - System Functions
  - Process Control
  - Importing Modules
  - Data marshalling support
  - Parsing arguments and building values
  - String conversion and formatting
  - Reflection
  - Codec registry and support functions
- Abstract Objects Layer
  - Object Protocol
• Number Protocol
• Sequence Protocol
• Mapping Protocol
• Iterator Protocol
• Old Buffer Protocol

• Concrete Objects Layer
  • Fundamental Objects
  • Numeric Objects
  • Sequence Objects
  • Mapping Objects
  • Other Objects

• Initialization, Finalization, and Threads
  • Initializing and finalizing the interpreter
  • Process-wide parameters
  • Thread State and the Global Interpreter Lock
  • Sub-interpreter support
  • Asynchronous Notifications
  • Profiling and Tracing
  • Advanced Debugger Support

• Memory Management
  • Overview
  • Memory Interface
  • Examples

• Object Implementation Support
  • Allocating Objects on the Heap
  • Common Object Structures
  • Type Objects
  • Number Object Structures
  • Mapping Object Structures
  • Sequence Object Structures
  • Buffer Object Structures
  • Supporting Cyclic Garbage Collection
Python Frequently Asked Questions

- General Python FAQ
- Programming FAQ
- Design and History FAQ
- Library and Extension FAQ
- Extending/Embedding FAQ
- Python on Windows FAQ
- Graphic User Interface FAQ
- “Why is Python Installed on my Computer?” FAQ
Python Documentation contents

- What's New in Python
  - What's New in Python 2.7
    - The Future for Python 2.x
    - Changes to the Handling of Deprecation Warnings
    - Python 3.1 Features
    - PEP 372: Adding an Ordered Dictionary to collections
    - PEP 378: FormatSpecifier for Thousands Separator
    - PEP 389: The argparse Module for Parsing Command Lines
    - PEP 391: Dictionary-Based Configuration For Logging
    - PEP 3106: Dictionary Views
    - PEP 3137: The memoryview Object
  - Other Language Changes
    - Interpreter Changes
    - Optimizations
  - New and Improved Modules
    - New module: importlib
    - New module: sysconfig
    - ttk: Themed Widgets for Tk
    - Updated module: unittest
    - Updated module: ElementTree 1.3
  - Build and C API Changes
    - Capsules
    - Port-Specific Changes: Windows
    - Port-Specific Changes: Mac OS X
    - Port-Specific Changes: FreeBSD
  - Other Changes and Fixes
  - Porting to Python 2.7
  - New Features Added to Python 2.7 Maintenance Releases
    - PEP 434: IDLE Enhancement Exception for All
Branches

- PEP 466: Network Security Enhancements for Python 2.7
- PEP 477: Backport ensurepip (PEP 453) to Python 2.7
  - Bootstrapping pip By Default
  - Documentation Changes
- PEP 476: Enabling certificate verification by default for stdlib http clients
- PEP 493: HTTPS verification migration tools for Python 2.7

Acknowledgements

What’s New in Python 2.6

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

- Other Language Changes
  - Optimizations
  - Interpreter Changes

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

- Deprecations and Removals

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

- Porting to Python 2.6

- Acknowledgements

- What’s New in Python 2.5
  - PEP 308: Conditional Expressions
  - PEP 309: Partial Function Application
  - PEP 314: Metadata for Python Software Packages v1.1
  - PEP 328: Absolute and Relative Imports
  - PEP 338: Executing Modules as Scripts
  - PEP 341: Unified `try/except/finally`
  - PEP 342: New Generator Features
  - PEP 343: The ‘with’ statement
    - Writing Context Managers
    - The `contextlib` module
  - PEP 352: Exceptions as New-Style Classes
  - PEP 353: Using `ssize_t` as the index type
  - PEP 357: The ‘__index__’ method
Other Language Changes
  - Interactive Interpreter Changes
  - Optimizations
New, Improved, and Removed Modules
  - The ctypes package
  - The ElementTree package
  - The hashlib package
  - The sqlite3 package
  - The wsgiref package
Build and C API Changes
  - Port-Specific Changes
Porting to Python 2.5
Acknowledgements
  - What's New in Python 2.4
    - PEP 218: Built-In Set Objects
    - PEP 237: Unifying Long Integers and Integers
    - PEP 289: Generator Expressions
    - PEP 292: Simpler String Substitutions
    - PEP 318: Decorators for Functions and Methods
    - PEP 322: Reverse Iteration
    - PEP 324: New subprocess Module
    - PEP 327: Decimal Data Type
      - Why is Decimal needed?
      - The `Decimal` type
      - The `Context` type
    - PEP 328: Multi-line Imports
    - PEP 331: Locale-Independent Float/String Conversions
Other Language Changes
  - Optimizations
New, Improved, and Deprecated Modules
  - cookielib
  - doctest
Build and C API Changes
Port-Specific Changes

- Porting to Python 2.4
- Acknowledgements

What’s New in Python 2.3

- PEP 218: A Standard Set Datatype
- PEP 255: Simple Generators
- PEP 263: Source Code Encodings
- PEP 273: Importing Modules from ZIP Archives
- PEP 277: Unicode file name support for Windows NT
- PEP 278: Universal Newline Support
- PEP 279: enumerate()
- PEP 282: The logging Package
- PEP 285: A Boolean Type
- PEP 293: Codec Error Handling Callbacks
- PEP 301: Package Index and Metadata for Distutils
- PEP 302: New Import Hooks
- PEP 305: Comma-separated Files
- PEP 307: Pickle Enhancements
- Extended Slices
- Other Language Changes
  - String Changes
  - Optimizations
- New, Improved, and Deprecated Modules
  - Date/Time Type
  - The optparse Module
- Pymalloc: A Specialized Object Allocator
- Build and C API Changes
  - Port-Specific Changes
- Other Changes and Fixes
- Porting to Python 2.3
- Acknowledgements

What’s New in Python 2.2

- Introduction
- PEPs 252 and 253: Type and Class Changes
Old and New Classes
- Descriptors
  - Multiple Inheritance: The Diamond Rule
  - Attribute Access
  - Related Links
- PEP 234: Iterators
- PEP 255: Simple Generators
- PEP 237: Unifying Long Integers and Integers
- PEP 238: Changing the Division Operator
- Unicode Changes
- PEP 227: Nested Scopes
- New and Improved Modules
- Interpreter Changes and Fixes
- Other Changes and Fixes
- Acknowledgements

- What’s New in Python 2.1
  - Introduction
  - PEP 227: Nested Scopes
  - PEP 236: __future__ Directives
  - PEP 207: Rich Comparisons
  - PEP 230: Warning Framework
  - PEP 229: New Build System
  - PEP 205: Weak References
  - PEP 232: Function Attributes
  - PEP 235: Importing Modules on Case-Insensitive Platforms
  - PEP 217: Interactive Display Hook
  - PEP 208: New Coercion Model
  - PEP 241: Metadata in Python Packages
  - New and Improved Modules
  - Other Changes and Fixes
  - Acknowledgements

- What’s New in Python 2.0
  - Introduction
• What About Python 1.6?
• New Development Process
• Unicode
• List Comprehensions
• Augmented Assignment
• String Methods
• Garbage Collection of Cycles
• Other Core Changes
  • Minor Language Changes
  • Changes to Built-in Functions
• Porting to 2.0
• Extending/Embedding Changes
• Distutils: Making Modules Easy to Install
• XML Modules
  • SAX2 Support
  • DOM Support
  • Relationship to PyXML
• Module changes
• New modules
• IDLE Improvements
• Deleted and Deprecated Modules
• Acknowledgements

The Python Tutorial
  • 1. Whetting Your Appetite
  • 2. Using the Python Interpreter
    • 2.1. Invoking the Interpreter
      • 2.1.1. Argument Passing
      • 2.1.2. Interactive Mode
    • 2.2. The Interpreter and Its Environment
      • 2.2.1. Source Code Encoding
  • 3. An Informal Introduction to Python
    • 3.1. Using Python as a Calculator
      • 3.1.1. Numbers
      • 3.1.2. Strings
3.1.3. Unicode Strings
3.1.4. Lists
3.2. First Steps Towards Programming
4. More Control Flow Tools
  4.1. if Statements
  4.2. for Statements
  4.3. The range() Function
  4.4. break and continue Statements, and else Clauses on Loops
  4.5. pass Statements
  4.6. Defining Functions
  4.7. More on Defining Functions
    4.7.1. Default Argument Values
    4.7.2. Keyword Arguments
    4.7.3. Arbitrary Argument Lists
    4.7.4. Unpacking Argument Lists
    4.7.5. Lambda Expressions
    4.7.6. Documentation Strings
  4.8. Intermezzo: Coding Style
5. Data Structures
  5.1. More on Lists
    5.1.1. Using Lists as Stacks
    5.1.2. Using Lists as Queues
    5.1.3. Functional Programming Tools
    5.1.4. List Comprehensions
      5.1.4.1. Nested List Comprehensions
  5.2. The del statement
  5.3. Tuples and Sequences
  5.4. Sets
  5.5. Dictionaries
  5.6. Looping Techniques
  5.7. More on Conditions
  5.8. Comparing Sequences and Other Types
6. Modules
   ■ 6.1. More on Modules
      ■ 6.1.1. Executing modules as scripts
      ■ 6.1.2. The Module Search Path
      ■ 6.1.3. “Compiled” Python files
   ■ 6.2. Standard Modules
   ■ 6.3. The dir() Function
   ■ 6.4. Packages
      ■ 6.4.1. Importing * From a Package
      ■ 6.4.2. Intra-package References
      ■ 6.4.3. Packages in Multiple Directories

7. Input and Output
   ■ 7.1. Fancier Output Formatting
      ■ 7.1.1. Old string formatting
   ■ 7.2. Reading and Writing Files
      ■ 7.2.1. Methods of File Objects
      ■ 7.2.2. Saving structured data with json

8. Errors and Exceptions
   ■ 8.1. Syntax Errors
   ■ 8.2. Exceptions
   ■ 8.3. Handling Exceptions
   ■ 8.4. Raising Exceptions
   ■ 8.5. User-defined Exceptions
   ■ 8.6. Defining Clean-up Actions
   ■ 8.7. Predefined Clean-up Actions

9. Classes
   ■ 9.1. A Word About Names and Objects
   ■ 9.2. Python Scopes and Namespaces
   ■ 9.3. A First Look at Classes
      ■ 9.3.1. Class Definition Syntax
      ■ 9.3.2. Class Objects
      ■ 9.3.3. Instance Objects
      ■ 9.3.4. Method Objects
      ■ 9.3.5. Class and Instance Variables
- 9.4. Random Remarks
- 9.5. Inheritance
  - 9.5.1. Multiple Inheritance
- 9.6. Private Variables and Class-local References
- 9.7. Odds and Ends
- 9.8. Exceptions Are Classes Too
- 9.9. Iterators
- 9.10. Generators
- 9.11. Generator Expressions
- 10. Brief Tour of the Standard Library
  - 10.1. Operating System Interface
  - 10.2. File Wildcards
  - 10.3. Command Line Arguments
  - 10.4. Error Output Redirection and Program Termination
  - 10.5. String Pattern Matching
  - 10.6. Mathematics
  - 10.7. Internet Access
  - 10.8. Dates and Times
  - 10.9. Data Compression
  - 10.10. Performance Measurement
  - 10.11. Quality Control
  - 10.12. Batteries Included
- 11. Brief Tour of the Standard Library – Part II
  - 11.1. Output Formatting
  - 11.2. Templating
  - 11.3. Working with Binary Data Record Layouts
  - 11.4. Multi-threading
  - 11.5. Logging
  - 11.6. Weak References
  - 11.7. Tools for Working with Lists
  - 11.8. Decimal Floating Point Arithmetic
- 12. What Now?
- 13. Interactive Input Editing and History Substitution
- 13.1. Line Editing
- 13.2. History Substitution
- 13.3. Key Bindings
- 13.4. Alternatives to the Interactive Interpreter
- 14. Floating Point Arithmetic: Issues and Limitations
  - 14.1. Representation Error
- 15. Appendix
  - 15.1. Interactive Mode
    - 15.1.1. Error Handling
    - 15.1.2. Executable Python Scripts
    - 15.1.3. The Interactive Startup File
    - 15.1.4. The Customization Modules

- Python Setup and Usage
  - 1. Command line and environment
    - 1.1. Command line
      - 1.1.1. Interface options
      - 1.1.2. Generic options
      - 1.1.3. Miscellaneous options
      - 1.1.4. Options you shouldn’t use
    - 1.2. Environment variables
      - 1.2.1. Debug-mode variables
  - 2. Using Python on Unix platforms
    - 2.1. Getting and installing the latest version of Python
      - 2.1.1. On Linux
      - 2.1.2. On FreeBSD and OpenBSD
      - 2.1.3. On OpenSolaris
    - 2.2. Building Python
    - 2.3. Python-related paths and files
    - 2.4. Miscellaneous
    - 2.5. Editors
  - 3. Using Python on Windows
    - 3.1. Installing Python
    - 3.2. Alternative bundles
    - 3.3. Configuring Python
3.3.1. Excursus: Setting environment variables
3.3.2. Finding the Python executable
3.3.3. Finding modules
3.3.4. Executing scripts
3.4. Additional modules
  3.4.1. PyWin32
  3.4.2. Py2exe
  3.4.3. WConio
3.5. Compiling Python on Windows
3.6. Other resources

4. Using Python on a Macintosh
  4.1. Getting and Installing MacPython
    4.1.1. How to run a Python script
    4.1.2. Running scripts with a GUI
    4.1.3. Configuration
  4.2. The IDE
  4.3. Installing Additional Python Packages
  4.4. GUI Programming on the Mac
  4.5. Distributing Python Applications on the Mac
  4.6. Other Resources

The Python Language Reference
  1. Introduction
    1.1. Alternate Implementations
    1.2. Notation
  2. Lexical analysis
    2.1. Line structure
      2.1.1. Logical lines
      2.1.2. Physical lines
      2.1.3. Comments
      2.1.4. Encoding declarations
      2.1.5. Explicit line joining
      2.1.6. Implicit line joining
      2.1.7. Blank lines
      2.1.8. Indentation
2.1.9. Whitespace between tokens

2.2. Other tokens

2.3. Identifiers and keywords
   2.3.1. Keywords
   2.3.2. Reserved classes of identifiers

2.4. Literals
   2.4.1. String literals
   2.4.2. String literal concatenation
   2.4.3. Numeric literals
   2.4.4. Integer and long integer literals
   2.4.5. Floating point literals
   2.4.6. Imaginary literals

2.5. Operators

2.6. Delimiters

3. Data model
   3.1. Objects, values and types
   3.2. The standard type hierarchy
   3.3. New-style and classic classes
   3.4. Special method names
      3.4.1. Basic customization
      3.4.2. Customizing attribute access
         3.4.2.1. More attribute access for new-style classes
         3.4.2.2. Implementing Descriptors
         3.4.2.3. Invoking Descriptors
         3.4.2.4. __slots__
      3.4.3. Customizing class creation
      3.4.4. Customizing instance and subclass checks
      3.4.5. Emulating callable objects
      3.4.6. Emulating container types
      3.4.7. Additional methods for emulation of sequence types
      3.4.8. Emulating numeric types
      3.4.9. Coercion rules
3.4.10. With Statement Context Managers
3.4.11. Special method lookup for old-style classes
3.4.12. Special method lookup for new-style classes

4. Execution model
   4.1. Naming and binding
      4.1.1. Interaction with dynamic features
   4.2. Exceptions

5. Expressions
   5.1. Arithmetic conversions
   5.2. Atoms
      5.2.1. Identifiers (Names)
      5.2.2. Literals
      5.2.3. Parenthesized forms
      5.2.4. List displays
      5.2.5. Displays for sets and dictionaries
      5.2.6. Generator expressions
      5.2.7. Dictionary displays
      5.2.8. Set displays
      5.2.9. String conversions
      5.2.10. Yield expressions
         5.2.10.1. Generator-iterator methods
   5.3. Primaries
      5.3.1. Attribute references
      5.3.2. Subscriptions
      5.3.3. Slicings
      5.3.4. Calls
   5.4. The power operator
   5.5. Unary arithmetic and bitwise operations
   5.6. Binary arithmetic operations
   5.7. Shifting operations
   5.8. Binary bitwise operations
   5.9. Comparisons
   5.10. Boolean operations
5.11. Conditional Expressions
5.12. Lambdas
5.13. Expression lists
5.14. Evaluation order
5.15. Operator precedence

6. Simple statements
   6.1. Expression statements
   6.2. Assignment statements
      6.2.1. Augmented assignment statements
   6.3. The assert statement
   6.4. The pass statement
   6.5. The del statement
   6.6. The print statement
   6.7. The return statement
   6.8. The yield statement
   6.9. The raise statement
   6.10. The break statement
   6.11. The continue statement
   6.12. The import statement
      6.12.1. Future statements
   6.13. The global statement
   6.14. The exec statement

7. Compound statements
   7.1. The if statement
   7.2. The while statement
   7.3. The for statement
   7.4. The try statement
   7.5. The with statement
   7.6. Function definitions
   7.7. Class definitions

8. Top-level components
   8.1. Complete Python programs
8.2. File input
8.3. Interactive input
8.4. Expression input
9. Full Grammar specification

The Python Standard Library
1. Introduction
2. Built-in Functions
3. Non-essential Built-in Functions
4. Built-in Constants
   4.1. Constants added by the site module
5. Built-in Types
   5.1. Truth Value Testing
   5.2. Boolean Operations — and, or, not
   5.3. Comparisons
   5.4. Numeric Types — int, float, long, complex
      5.4.1. Bitwise Operations on Integer Types
      5.4.2. Additional Methods on Integer Types
      5.4.3. Additional Methods on Float
   5.5. Iterator Types
      5.5.1. Generator Types
   5.6. Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange
      5.6.1. String Methods
      5.6.2. String Formatting Operations
      5.6.3. XRange Type
      5.6.4. Mutable Sequence Types
   5.7. Set Types — set, frozenset
   5.8. Mapping Types — dict
      5.8.1. Dictionary view objects
   5.9. File Objects
   5.10. memoryview type
   5.11. Context Manager Types
   5.12. Other Built-in Types
- 5.12.1. Modules
- 5.12.2. Classes and Class Instances
- 5.12.3. Functions
- 5.12.4. Methods
- 5.12.5. Code Objects
- 5.12.6. Type Objects
- 5.12.7. The Null Object
- 5.12.8. The Ellipsis Object
- 5.12.9. The Not Implemented Object
- 5.12.10. Boolean Values
- 5.12.11. Internal Objects
- 5.13. Special Attributes
- 6. Built-in Exceptions
  - 6.1. Exception hierarchy
- 7. String Services
  - 7.1. string — Common string operations
    - 7.1.1. String constants
    - 7.1.2. Custom String Formatting
    - 7.1.3. Format String Syntax
      - 7.1.3.1. Format Specification Mini-Language
      - 7.1.3.2. Format examples
    - 7.1.4. Template strings
    - 7.1.5. String functions
    - 7.1.6. Deprecated string functions
  - 7.2. re — Regular expression operations
    - 7.2.1. Regular Expression Syntax
    - 7.2.2. Module Contents
    - 7.2.3. Regular Expression Objects
    - 7.2.4. Match Objects
    - 7.2.5. Examples
      - 7.2.5.1. Checking For a Pair
      - 7.2.5.2. Simulating scanf()
      - 7.2.5.3. search() vs. match()
      - 7.2.5.4. Making a Phonebook
- 7.2.5.5. Text Munging
- 7.2.5.6. Finding all Adverbs
- 7.2.5.7. Finding all Adverbs and their Positions
- 7.2.5.8. Raw String Notation
- 7.3. `struct` — Interpret strings as packed binary data
  - 7.3.1. Functions and Exceptions
  - 7.3.2. Format Strings
    - 7.3.2.1. Byte Order, Size, and Alignment
    - 7.3.2.2. Format Characters
    - 7.3.2.3. Examples
  - 7.3.3. Classes
- 7.4. `difflib` — Helpers for computing deltas
  - 7.4.1. SequenceMatcher Objects
  - 7.4.2. SequenceMatcher Examples
  - 7.4.3. Differ Objects
  - 7.4.4. Differ Example
  - 7.4.5. A command-line interface to difflib
- 7.5. `StringIO` — Read and write strings as files
- 7.6. `cStringIO` — Faster version of `StringIO`
- 7.7. `textwrap` — Text wrapping and filling
- 7.8. `codecs` — Codec registry and base classes
  - 7.8.1. Codec Base Classes
    - 7.8.1.1. Codec Objects
    - 7.8.1.2. IncrementalEncoder Objects
    - 7.8.1.3. IncrementalDecoder Objects
    - 7.8.1.4. StreamWriter Objects
    - 7.8.1.5. StreamReader Objects
    - 7.8.1.6. StreamReaderWriter Objects
    - 7.8.1.7. StreamRecoder Objects
  - 7.8.2. Encodings and Unicode
  - 7.8.3. Standard Encodings
  - 7.8.4. Python Specific Encodings
- 7.8.5. **encodings.idna** — Internationalized Domain Names in Applications
- 7.8.6. **encodings.utf_8_sig** — UTF-8 codec with BOM signature
- 7.9. **unicodedata** — Unicode Database
- 7.10. **stringprep** — Internet String Preparation
- 7.11. **fpformat** — Floating point conversions
- 8. Data Types
  - 8.1. **datetime** — Basic date and time types
    - 8.1.1. Available Types
    - 8.1.2. **timedelta** Objects
    - 8.1.3. **date** Objects
    - 8.1.4. **datetime** Objects
    - 8.1.5. **time** Objects
    - 8.1.6. **tzinfo** Objects
    - 8.1.7. **strftime()** and **strptime()** Behavior
  - 8.2. **calendar** — General calendar-related functions
  - 8.3. **collections** — High-performance container datatypes
    - 8.3.1. **Counter** objects
    - 8.3.2. **deque** objects
      - 8.3.2.1. **deque** Recipes
    - 8.3.3. **defaultdict** objects
      - 8.3.3.1. **defaultdict** Examples
    - 8.3.4. **namedtuple()** Factory Function for Tuples with Named Fields
    - 8.3.5. **OrderedDict** objects
      - 8.3.5.1. **OrderedDict** Examples and Recipes
    - 8.3.6. Collections Abstract Base Classes
  - 8.4. **heapq** — Heap queue algorithm
    - 8.4.1. Basic Examples
8.4.2. Priority Queue Implementation Notes
8.4.3. Theory
8.5. bisect — Array bisection algorithm
  8.5.1. Searching Sorted Lists
  8.5.2. Other Examples
8.6. array — Efficient arrays of numeric values
8.7. sets — Unordered collections of unique elements
  8.7.1. Set Objects
  8.7.2. Example
  8.7.3. Protocol for automatic conversion to immutable
  8.7.4. Comparison to the built-in set types
8.8. sched — Event scheduler
  8.8.1. Scheduler Objects
8.9. mutex — Mutual exclusion support
  8.9.1. Mutex Objects
8.10. Queue — A synchronized queue class
  8.10.1. Queue Objects
8.11. weakref — Weak references
  8.11.1. Weak Reference Objects
  8.11.2. Example
8.12. UserDict — Class wrapper for dictionary objects
8.13. UserList — Class wrapper for list objects
8.14. UserString — Class wrapper for string objects
8.15. types — Names for built-in types
8.16. new — Creation of runtime internal objects
8.17. copy — Shallow and deep copy operations
8.18. pprint — Data pretty printer
  8.18.1. PrettyPrinter Objects
  8.18.2. pprint Example
8.19. repr — Alternate repr() implementation
  8.19.1. Repr Objects
- 8.19.2. Subclassing Repr Objects
- 9. Numeric and Mathematical Modules
  - 9.1. numbers — Numeric abstract base classes
    - 9.1.1. The numeric tower
    - 9.1.2. Notes for type implementors
      - 9.1.2.1. Adding More Numeric ABCs
      - 9.1.2.2. Implementing the arithmetic operations
  - 9.2. math — Mathematical functions
    - 9.2.1. Number-theoretic and representation functions
    - 9.2.2. Power and logarithmic functions
    - 9.2.3. Trigonometric functions
    - 9.2.4. Angular conversion
    - 9.2.5. Hyperbolic functions
    - 9.2.6. Special functions
    - 9.2.7. Constants
  - 9.3. cmath — Mathematical functions for complex numbers
    - 9.3.1. Conversions to and from polar coordinates
    - 9.3.2. Power and logarithmic functions
    - 9.3.3. Trigonometric functions
    - 9.3.4. Hyperbolic functions
    - 9.3.5. Classification functions
    - 9.3.6. Constants
  - 9.4. decimal — Decimal fixed point and floating point arithmetic
    - 9.4.1. Quick-start Tutorial
    - 9.4.2. Decimal objects
      - 9.4.2.1. Logical operands
    - 9.4.3. Context objects
    - 9.4.4. Signals
    - 9.4.5. Floating Point Notes
      - 9.4.5.1. Mitigating round-off error with
increased precision

-  9.4.5.2. Special values
-  9.4.6. Working with threads
-  9.4.7. Recipes
-  9.4.8. Decimal FAQ

-  9.5. fractions — Rational numbers
-  9.6. random — Generate pseudo-random numbers
-  9.7. itertools — Functions creating iterators for efficient looping
  -  9.7.1. Itertool functions
  -  9.7.2. Recipes
-  9.8. functools — Higher-order functions and operations on callable objects
  -  9.8.1. partial Objects
-  9.9. operator — Standard operators as functions
  -  9.9.1. Mapping Operators to Functions

-  10. File and Directory Access
  -  10.1. os.path — Common pathname manipulations
  -  10.2. fileinput — Iterate over lines from multiple input streams
  -  10.3. stat — Interpreting stat() results
  -  10.4. statvfs — Constants used with os.statvfs()
  -  10.5. filecmp — File and Directory Comparisons
    -  10.5.1. The dircmp class
  -  10.6. tempfile — Generate temporary files and directories
  -  10.7. glob — Unix style pathname pattern expansion
  -  10.8. fnmatch — Unix filename pattern matching
  -  10.9. linecache — Random access to text lines
  -  10.10. shutil — High-level file operations
    -  10.10.1. Directory and files operations
      -  10.10.1.1. copytree example
- 10.10.2. Archiving operations
  - 10.10.2.1. Archiving example
- 10.11. dircache — Cached directory listings
- 10.12. macpath — Mac OS 9 path manipulation functions
- 11. Data Persistence
  - 11.1. pickle — Python object serialization
    - 11.1.1. Relationship to other Python modules
    - 11.1.2. Data stream format
    - 11.1.3. Usage
    - 11.1.4. What can be pickled and unpickled?
    - 11.1.5. The pickle protocol
      - 11.1.5.1. Pickling and unpickling normal class instances
      - 11.1.5.2. Pickling and unpickling extension types
      - 11.1.5.3. Pickling and unpickling external objects
    - 11.1.6. Subclassing Unpicklers
    - 11.1.7. Example
  - 11.2. cPickle — A faster pickle
  - 11.3. copy_reg — Register pickle support functions
    - 11.3.1. Example
  - 11.4. shelve — Python object persistence
    - 11.4.1. Restrictions
    - 11.4.2. Example
  - 11.5. marshal — Internal Python object serialization
  - 11.6. anydbm — Generic access to DBM-style databases
  - 11.7. whichdb — Guess which DBM module created a database
  - 11.8. dbm — Simple “database” interface
  - 11.9. gdbm — GNU’s reinterpretation of dbm
- 11.10. dbhash — DBM-style interface to the BSD database library
  - 11.10.1. Database Objects

- 11.11. bsddb — Interface to Berkeley DB library
  - 11.11.1. Hash, BTree and Record Objects

- 11.12. dumbdbm — Portable DBM implementation
  - 11.12.1. Dumbdbm Objects

- 11.13. sqlite3 — DB-API 2.0 interface for SQLite databases
  - 11.13.1. Module functions and constants
  - 11.13.2. Connection Objects
  - 11.13.3. Cursor Objects
  - 11.13.4. Row Objects
  - 11.13.5. SQLite and Python types
    - 11.13.5.1. Introduction
    - 11.13.5.2. Using adapters to store additional Python types in SQLite databases
      - 11.13.5.2.1. Letting your object adapt itself
      - 11.13.5.2.2. Registering an adapter callable
    - 11.13.5.3. Converting SQLite values to custom Python types
    - 11.13.5.4. Default adapters and converters
  - 11.13.6. Controlling Transactions
  - 11.13.7. Using sqlite3 efficiently
    - 11.13.7.1. Using shortcut methods
    - 11.13.7.2. Accessing columns by name instead of by index
    - 11.13.7.3. Using the connection as a context manager
  - 11.13.8. Common issues
    - 11.13.8.1. Multithreading

- 12. Data Compression and Archiving
12.1. **zlib** — Compression compatible with **gzip**

12.2. **gzip** — Support for **gzip** files
   - 12.2.1. Examples of usage

12.3. **bz2** — Compression compatible with **bzip2**
   - 12.3.1. (De)compression of files
   - 12.3.2. Sequential (de)compression
   - 12.3.3. One-shot (de)compression

12.4. **zipfile** — Work with ZIP archives
   - 12.4.1. ZipFile Objects
   - 12.4.2. PyZipFile Objects
   - 12.4.3. ZipInfo Objects
   - 12.4.4. Command-Line Interface
     - 12.4.4.1. Command-line options

12.5. **tarfile** — Read and write tar archive files
   - 12.5.1. TarFile Objects
   - 12.5.2. TarInfo Objects
   - 12.5.3. Examples
   - 12.5.4. Supported tar formats
   - 12.5.5. Unicode issues

13. File Formats
   - 13.1. **csv** — CSV File Reading and Writing
     - 13.1.1. Module Contents
     - 13.1.2. Dialects and Formatting Parameters
     - 13.1.3. Reader Objects
     - 13.1.4. Writer Objects
     - 13.1.5. Examples

13.2. **ConfigParser** — Configuration file parser
   - 13.2.1. RawConfigParser Objects
   - 13.2.2. ConfigParser Objects
   - 13.2.3. SafeConfigParser Objects
   - 13.2.4. Examples

13.3. **robotparser** — Parser for robots.txt

13.4. **netrc** — netrc file processing
13.4.1. netrc Objects

13.5. xdrlib — Encode and decode XDR data
   13.5.1. Packer Objects
   13.5.2. Unpacker Objects
   13.5.3. Exceptions

13.6. plistlib — Generate and parse Mac OS X .plist files
   13.6.1. Examples

14. Cryptographic Services
   14.1. hashlib — Secure hashes and message digests
      14.1.1. Key derivation
   14.2. hmac — Keyed-Hashing for Message Authentication
   14.3. md5 — MD5 message digest algorithm
   14.4. sha — SHA-1 message digest algorithm

15. Generic Operating System Services
   15.1. os — Miscellaneous operating system interfaces
      15.1.1. Process Parameters
      15.1.2. File Object Creation
      15.1.3. File Descriptor Operations
         15.1.3.1. open() flag constants
      15.1.4. Files and Directories
      15.1.5. Process Management
      15.1.6. Miscellaneous System Information
      15.1.7. Miscellaneous Functions
   15.2. io — Core tools for working with streams
      15.2.1. Module Interface
      15.2.2. I/O Base Classes
      15.2.3. Raw File I/O
      15.2.4. Buffered Streams
      15.2.5. Text I/O
      15.2.6. Advanced topics
         15.2.6.1. Performance
- 15.2.6.1.1. Binary I/O
- 15.2.6.1.2. Text I/O
  - 15.2.6.2. Multi-threading
  - 15.2.6.3. Reentrancy
- 15.3. time — Time access and conversions
- 15.4. argparse — Parser for command-line options, arguments and sub-commands
  - 15.4.1. Example
    - 15.4.1.1. Creating a parser
    - 15.4.1.2. Adding arguments
    - 15.4.1.3. Parsing arguments
  - 15.4.2. ArgumentParser objects
    - 15.4.2.1. prog
    - 15.4.2.2. usage
    - 15.4.2.3. description
    - 15.4.2.4. epilog
    - 15.4.2.5. parents
    - 15.4.2.6. formatter_class
    - 15.4.2.7. prefix_chars
    - 15.4.2.8. fromfile_prefix_chars
    - 15.4.2.9. argument_default
    - 15.4.2.10. conflict_handler
    - 15.4.2.11. add_help
  - 15.4.3. The add_argument() method
    - 15.4.3.1. name or flags
    - 15.4.3.2. action
    - 15.4.3.3. nargs
    - 15.4.3.4. const
    - 15.4.3.5. default
    - 15.4.3.6. type
    - 15.4.3.7. choices
    - 15.4.3.8. required
    - 15.4.3.9. help
    - 15.4.3.10. metavar
15.4.3.11. dest
15.4.3.12. Action classes

15.4.4. The parse_args() method
15.4.4.1. Option value syntax
15.4.4.2. Invalid arguments
15.4.4.3. Arguments containing -
15.4.4.4. Argument abbreviations (prefix matching)
15.4.4.5. Beyond sys.argv
15.4.4.6. The Namespace object

15.4.5. Other utilities
15.4.5.1. Sub-commands
15.4.5.2. FileType objects
15.4.5.3. Argument groups
15.4.5.4. Mutual exclusion
15.4.5.5. Parser defaults
15.4.5.6. Printing help
15.4.5.7. Partial parsing
15.4.5.8. Customizing file parsing
15.4.5.9. Exiting methods

15.4.6. Upgrading optparse code

15.5. optparse — Parser for command line options

15.5.1. Background
15.5.1.1. Terminology
15.5.1.2. What are options for?
15.5.1.3. What are positional arguments for?

15.5.2. Tutorial
15.5.2.1. Understanding option actions
15.5.2.2. The store action
15.5.2.3. Handling boolean (flag) options
15.5.2.4. Other actions
15.5.2.5. Default values
15.5.2.6. Generating help
15.5.2.6.1. Grouping Options
15.5.2.7. Printing a version string
15.5.2.8. How optparse handles errors
15.5.2.9. Putting it all together

15.5.3. Reference Guide
15.5.3.1. Creating the parser
15.5.3.2. Populating the parser
15.5.3.3. Defining options
15.5.3.4. Option attributes
15.5.3.5. Standard option actions
15.5.3.6. Standard option types
15.5.3.7. Parsing arguments
15.5.3.8. Querying and manipulating your option parser
15.5.3.9. Conflicts between options
15.5.3.10. Cleanup
15.5.3.11. Other methods

15.5.4. Option Callbacks
15.5.4.1. Defining a callback option
15.5.4.2. How callbacks are called
15.5.4.3. Raising errors in a callback
15.5.4.4. Callback example 1: trivial callback
15.5.4.5. Callback example 2: check option order
15.5.4.6. Callback example 3: check option order (generalized)
15.5.4.7. Callback example 4: check arbitrary condition
15.5.4.8. Callback example 5: fixed arguments
15.5.4.9. Callback example 6: variable arguments

15.5.5. Extending optparse
15.5.5.1. Adding new types
15.5.5.2. Adding new actions
15.6. **getopt** — C-style parser for command line options

15.7. **logging** — Logging facility for Python

- 15.7.1. Logger Objects
- 15.7.2. Logging Levels
- 15.7.3. Handler Objects
- 15.7.4. Formatter Objects
- 15.7.5. Filter Objects
- 15.7.6. LogRecord Objects
- 15.7.7. LogRecord attributes
- 15.7.8. LoggerAdapter Objects
- 15.7.9. Thread Safety
- 15.7.10. Module-Level Functions
- 15.7.11. Integration with the warnings module

15.8. **logging.config** — Logging configuration

- 15.8.1. Configuration functions
- 15.8.2. Configuration dictionary schema
  - 15.8.2.1. Dictionary Schema Details
  - 15.8.2.2. Incremental Configuration
  - 15.8.2.3. Object connections
  - 15.8.2.4. User-defined objects
  - 15.8.2.5. Access to external objects
  - 15.8.2.6. Access to internal objects
  - 15.8.2.7. Import resolution and custom importers
- 15.8.3. Configuration file format

15.9. **logging.handlers** — Logging handlers

- 15.9.1. StreamHandler
- 15.9.2. FileHandler
- 15.9.3. NullHandler
- 15.9.4. WatchedFileHandler
- 15.9.5. RotatingFileHandler
- 15.9.6. TimedRotatingFileHandler
- 15.9.7. SocketHandler
- 15.9.8. DatagramHandler
- 15.9.9. SysLogHandler
- 15.9.10. NTEventLogHandler
- 15.9.11. SMTPHandler
- 15.9.12. MemoryHandler
- 15.9.13. HTTPHandler
- 15.10. getpass — Portable password input
- 15.11. curses — Terminal handling for character-cell displays
  - 15.11.1. Functions
  - 15.11.2. Window Objects
  - 15.11.3. Constants
- 15.12. curses.textpad — Text input widget for curses programs
  - 15.12.1. Textbox objects
- 15.13. curses.ascii — Utilities for ASCII characters
- 15.14. curses.panel — A panel stack extension for curses
  - 15.14.1. Functions
  - 15.14.2. Panel Objects
- 15.15. platform — Access to underlying platform's identifying data
  - 15.15.1. Cross Platform
  - 15.15.2. Java Platform
  - 15.15.3. Windows Platform
    - 15.15.3.1. Win95/98 specific
  - 15.15.4. Mac OS Platform
  - 15.15.5. Unix Platforms
- 15.16. errno — Standard errno system symbols
- 15.17. ctypes — A foreign function library for Python
  - 15.17.1. ctypes tutorial
    - 15.17.1.1. Loading dynamic link libraries
    - 15.17.1.2. Accessing functions from loaded
15.17.1.3. Calling functions
15.17.1.4. Fundamental data types
15.17.1.5. Calling functions, continued
15.17.1.6. Calling functions with your own custom data types
15.17.1.7. Specifying the required argument types (function prototypes)
15.17.1.8. Return types
15.17.1.9. Passing pointers (or: passing parameters by reference)
15.17.1.10. Structures and unions
15.17.1.11. Structure/union alignment and byte order
15.17.1.12. Bit fields in structures and unions
15.17.1.13. Arrays
15.17.1.14. Pointers
15.17.1.15. Type conversions
15.17.1.16. Incomplete Types
15.17.1.17. Callback functions
15.17.1.18. Accessing values exported from dlls
15.17.1.19. Surprises
15.17.1.20. Variable-sized data types

15.17.2. ctypes reference
15.17.2.1. Finding shared libraries
15.17.2.2. Loading shared libraries
15.17.2.3. Foreign functions
15.17.2.4. Function prototypes
15.17.2.5. Utility functions
15.17.2.6. Data types
15.17.2.7. Fundamental data types
15.17.2.8. Structured data types
15.17.2.9. Arrays and pointers
16. Optional Operating System Services
   16.1. select — Waiting for I/O completion
       16.1.1. Edge and Level Trigger Polling (epoll) Objects
       16.1.2. Polling Objects
       16.1.3. Kqueue Objects
       16.1.4. Kevent Objects
   16.2. threading — Higher-level threading interface
       16.2.1. Thread Objects
       16.2.2. Lock Objects
       16.2.3. RLock Objects
       16.2.4. Condition Objects
       16.2.5. Semaphore Objects
           16.2.5.1. Semaphore Example
       16.2.6. Event Objects
       16.2.7. Timer Objects
       16.2.8. Using locks, conditions, and semaphores in the with statement
       16.2.9. Importing in threaded code
   16.3. thread — Multiple threads of control
   16.4. dummy_threading — Drop-in replacement for the threading module
   16.5. dummy_thread — Drop-in replacement for the thread module
   16.6. multiprocessing — Process-based “threading” interface
       16.6.1. Introduction
           16.6.1.1. The Process class
           16.6.1.2. Exchanging objects between processes
           16.6.1.3. Synchronization between processes
           16.6.1.4. Sharing state between processes
           16.6.1.5. Using a pool of workers
16.6.2. Reference
  16.6.2.1. Process and exceptions
  16.6.2.2. Pipes and Queues
  16.6.2.3. Miscellaneous
  16.6.2.4. Connection Objects
  16.6.2.5. Synchronization primitives
  16.6.2.6. Shared ctypes Objects
    16.6.2.6.1. The multiprocessing.sharedctypes module
  16.6.2.7. Managers
    16.6.2.7.1. Customized managers
    16.6.2.7.2. Using a remote manager
  16.6.2.8. Proxy Objects
    16.6.2.8.1. Cleanup
  16.6.2.9. Process Pools
  16.6.2.10. Listeners and Clients
    16.6.2.10.1. Address Formats
  16.6.2.11. Authentication keys
  16.6.2.12. Logging
  16.6.2.13. The multiprocessing dummy module
16.6.3. Programming guidelines
  16.6.3.1. All platforms
  16.6.3.2. Windows
16.6.4. Examples
16.7. mmap — Memory-mapped file support
16.8. readline — GNU readline interface
  16.8.1. Init file
  16.8.2. Line buffer
  16.8.3. History file
  16.8.4. History list
  16.8.5. Startup hooks
  16.8.6. Completion
  16.8.7. Example
16.9. r1completer — Completion function for GNU readline
   16.9.1. Completer Objects

17. Interprocess Communication and Networking
   17.1. subprocess — Subprocess management
      17.1.1. Using the subprocess Module
         17.1.1.1. Frequently Used Arguments
         17.1.1.2. Popen Constructor
         17.1.1.3. Exceptions
         17.1.1.4. Security
      17.1.2. Popen Objects
      17.1.3. Windows Popen Helpers
         17.1.3.1. Constants
      17.1.4. Replacing Older Functions with the subprocess Module
         17.1.4.1. Replacing /bin/sh shell backquote
         17.1.4.2. Replacing shell pipeline
         17.1.4.3. Replacing os.system()
         17.1.4.4. Replacing the os.spawn family
         17.1.4.5. Replacing os.popen(), os.popen2(), os.popen3()
         17.1.4.6. Replacing functions from the open2 module
      17.1.5. Notes
         17.1.5.1. Converting an argument sequence to a string on Windows
   17.2. socket — Low-level networking interface
      17.2.1. Socket Objects
      17.2.2. Example
   17.3. ssl — TLS/SSL wrapper for socket objects
      17.3.1. Functions, Constants, and Exceptions
         17.3.1.1. Socket creation
         17.3.1.2. Context creation
- 17.3.1.3. Random generation
- 17.3.1.4. Certificate handling
- 17.3.1.5. Constants
- 17.3.2. SSL Sockets
- 17.3.3. SSL Contexts
- 17.3.4. Certificates
  - 17.3.4.1. Certificate chains
  - 17.3.4.2. CA certificates
  - 17.3.4.3. Combined key and certificate
  - 17.3.4.4. Self-signed certificates
- 17.3.5. Examples
  - 17.3.5.1. Testing for SSL support
  - 17.3.5.2. Client-side operation
  - 17.3.5.3. Server-side operation
- 17.3.6. Notes on non-blocking sockets
- 17.3.7. Security considerations
  - 17.3.7.1. Best defaults
  - 17.3.7.2. Manual settings
    - 17.3.7.2.1. Verifying certificates
    - 17.3.7.2.2. Protocol versions
    - 17.3.7.2.3. Cipher selection
  - 17.3.7.3. Multi-processing
- 17.4. signal — Set handlers for asynchronous events
  - 17.4.1. Example
- 17.5. popen2 — Subprocesses with accessible I/O streams
  - 17.5.1. Popen3 and Popen4 Objects
  - 17.5.2. Flow Control Issues
- 17.6. asyncore — Asynchronous socket handler
  - 17.6.1. asyncore Example basic HTTP client
  - 17.6.2. asyncore Example basic echo server
- 17.7. asynchat — Asynchronous socket command/response handler
  - 17.7.1. asynchat - Auxiliary Classes
17.7.2. asynchat Example

18. Internet Data Handling
- 18.1. email — An email and MIME handling package
  - 18.1.1. email.message: Representing an email message
  - 18.1.2. email.parser: Parsing email messages
    - 18.1.2.1. FeedParser API
    - 18.1.2.2. Parser class API
    - 18.1.2.3. Additional notes
  - 18.1.3. email.generator: Generating MIME documents
  - 18.1.4. email.mime: Creating email and MIME objects from scratch
  - 18.1.5. email.header: Internationalized headers
  - 18.1.6. email.charset: Representing character sets
  - 18.1.7. email.encoders: Encoders
  - 18.1.8. email.errors: Exception and Defect classes
  - 18.1.9. email.utils: Miscellaneous utilities
  - 18.1.10. email.iterators: Iterators
  - 18.1.11. email: Examples
  - 18.1.12. Package History
  - 18.1.13. Differences from mimelib
- 18.2. json — JSON encoder and decoder
  - 18.2.1. Basic Usage
  - 18.2.2. Encoders and Decoders
  - 18.2.3. Standard Compliance and Interoperability
    - 18.2.3.1. Character Encodings
    - 18.2.3.2. Infinite and NaN Number Values
    - 18.2.3.3. Repeated Names Within an Object
    - 18.2.3.4. Top-level Non-Object, Non-Array Values
18.2.3.5. Implementation Limitations

18.3. mailcap — Mailcap file handling

18.4. mailbox — Manipulate mailboxes in various formats
  - 18.4.1. Mailbox objects
    - 18.4.1.1. Maildir
    - 18.4.1.2. mbox
    - 18.4.1.3. MH
    - 18.4.1.4. Babyl
    - 18.4.1.5. MMDF
  - 18.4.2. Message objects
    - 18.4.2.1. MaildirMessage
    - 18.4.2.2. mboxMessage
    - 18.4.2.3. MHMessage
    - 18.4.2.4. BabylMessage
    - 18.4.2.5. MMDFMessage
  - 18.4.3. Exceptions
  - 18.4.4. Deprecated classes and methods
  - 18.4.5. Examples

18.5. mhlib — Access to MH mailboxes
  - 18.5.1. MH Objects
  - 18.5.2. Folder Objects
  - 18.5.3. Message Objects

18.6. mimeutils — Tools for parsing MIME messages
  - 18.6.1. Additional Methods of Message Objects

18.7. mimetypes — Map filenames to MIME types
  - 18.7.1. MimeTypes Objects

18.8. MimeWriter — Generic MIME file writer
  - 18.8.1. MimeWriter Objects

18.9. mimify — MIME processing of mail messages

18.10. multifile — Support for files containing distinct parts
18.10.1. MultiFile Objects
18.10.2. MultiFile Example

18.11. rfc822 — Parse RFC 2822 mail headers
   18.11.1. Message Objects
   18.11.2. AddressList Objects

18.12. base64 — RFC 3548: Base16, Base32, Base64 Data Encodings

18.13. binhex — Encode and decode binhex4 files
   18.13.1. Notes

18.14. binascii — Convert between binary and ASCII

18.15. quopri — Encode and decode MIME quoted-printable data

18.16. uu — Encode and decode uuencode files

19. Structured Markup Processing Tools
   19.1. HTMLParser — Simple HTML and XHTML parser
      19.1.1. Example HTML Parser Application
      19.1.2. HTMLParser Methods
      19.1.3. Examples
   19.2. sgmllib — Simple SGML parser
   19.3. htmllib — A parser for HTML documents
      19.3.1. HTMLParser Objects
   19.4. htmlentitydefs — Definitions of HTML general entities
   19.5. XML Processing Modules
   19.6. XML vulnerabilities
      19.6.1. defused packages
   19.7. xml.etree.ElementTree — The ElementTree XML API
      19.7.1. Tutorial
         19.7.1.1. XML tree and elements
         19.7.1.2. Parsing XML
         19.7.1.3. Finding interesting elements
         19.7.1.4. Modifying an XML File
- 19.7.1.5. Building XML documents
- 19.7.1.6. Parsing XML with Namespaces
- 19.7.1.7. Additional resources
- 19.7.2. XPath support
  - 19.7.2.1. Example
  - 19.7.2.2. Supported XPath syntax
- 19.7.3. Reference
  - 19.7.3.1. Functions
  - 19.7.3.2. Element Objects
  - 19.7.3.3. ElementTree Objects
  - 19.7.3.4. QName Objects
  - 19.7.3.5. TreeBuilder Objects
  - 19.7.3.6. XMLParser Objects
- 19.8. `xml.dom` — The Document Object Model API
  - 19.8.1. Module Contents
  - 19.8.2. Objects in the DOM
    - 19.8.2.1. DOMImplementation Objects
    - 19.8.2.2. Node Objects
    - 19.8.2.3. NodeList Objects
    - 19.8.2.4. DocumentType Objects
    - 19.8.2.5. Document Objects
    - 19.8.2.6. Element Objects
    - 19.8.2.7. Attr Objects
    - 19.8.2.8. NamedNodeMap Objects
    - 19.8.2.9. Comment Objects
    - 19.8.2.10. Text and CDATASection Objects
    - 19.8.2.11. ProcessingInstruction Objects
    - 19.8.2.12. Exceptions
  - 19.8.3. Conformance
    - 19.8.3.1. Type Mapping
    - 19.8.3.2. Accessor Methods
- 19.9. `xml.dom.minidom` — Minimal DOM implementation
  - 19.9.1. DOM Objects
  - 19.9.2. DOM Example
- 19.9.3. minidom and the DOM standard
- 19.10. xml.dom.pulldom — Support for building partial DOM trees
  - 19.10.1. DOMEventStream Objects
- 19.11. xml.sax — Support for SAX2 parsers
  - 19.11.1. SAXException Objects
- 19.12. xml.sax.handler — Base classes for SAX handlers
  - 19.12.1. ContentHandler Objects
  - 19.12.2. DTDHandler Objects
  - 19.12.3. EntityResolver Objects
  - 19.12.4. ErrorHandler Objects
- 19.13. xml.sax.saxutils — SAX Utilities
  - 19.14.1. XMLReader Objects
  - 19.14.2. IncrementalParser Objects
  - 19.14.3. Locator Objects
  - 19.14.4. InputSource Objects
  - 19.14.5. The Attributes Interface
  - 19.14.6. The AttributesNS Interface
- 19.15. xml.parsers.expat — Fast XML parsing using Expat
  - 19.15.1. XMLParser Objects
  - 19.15.2. ExpatError Exceptions
  - 19.15.3. Example
  - 19.15.4. Content Model Descriptions
  - 19.15.5. Expat error constants

- 20. Internet Protocols and Support
  - 20.1. webbrowser — Convenient Web-browser controller
    - 20.1.1. Browser Controller Objects
  - 20.2. cgi — Common Gateway Interface support
    - 20.2.1. Introduction
20.2.2. Using the cgi module
20.2.3. Higher Level Interface
20.2.4. Old classes
20.2.5. Functions
20.2.6. Caring about security
20.2.7. Installing your CGI script on a Unix system
20.2.8. Testing your CGI script
20.2.9. Debugging CGI scripts
20.2.10. Common problems and solutions

20.3. cgitb — Traceback manager for CGI scripts

20.4. wsgiref — WSGI Utilities and Reference Implementation
   20.4.1. wsgiref.util — WSGI environment utilities
   20.4.2. wsgiref.headers — WSGI response header tools
   20.4.3. wsgiref.simple_server — a simple WSGI HTTP server
   20.4.4. wsgiref.validate — WSGI conformance checker
   20.4.5. wsgiref.handlers — server/gateway base classes
   20.4.6. Examples

20.5. urllib — Open arbitrary resources by URL
   20.5.1. High-level interface
   20.5.2. Utility functions
   20.5.3. URL Opener objects
   20.5.4. urllib Restrictions
   20.5.5. Examples

20.6. urllib2 — extensible library for opening URLs
   20.6.1. Request Objects
   20.6.2. OpenerDirector Objects
   20.6.3. BaseHandler Objects
   20.6.4. HTTPRedirectHandler Objects
- 20.6.5. HTTPCookieProcessor Objects
- 20.6.6. ProxyHandler Objects
- 20.6.7. HTTPPasswordMgr Objects
- 20.6.8. AbstractBasicAuthHandler Objects
- 20.6.9. HTTPBasicAuthHandler Objects
- 20.6.10. ProxyBasicAuthHandler Objects
- 20.6.11. AbstractDigestAuthHandler Objects
- 20.6.12. HTTPDigestAuthHandler Objects
- 20.6.13. ProxyDigestAuthHandler Objects
- 20.6.14. HTTPHandler Objects
- 20.6.15. HTTPSHandler Objects
- 20.6.16. FileHandler Objects
- 20.6.17. FTPHandler Objects
- 20.6.18. CacheFTPHandler Objects
- 20.6.19. UnknownHandler Objects
- 20.6.20. HTTPErrorProcessor Objects
- 20.6.21. Examples

- 20.7. httpplib — HTTP protocol client
  - 20.7.1. HTTPConnection Objects
  - 20.7.2. HTTPResponse Objects
  - 20.7.3. Examples

- 20.8. ftplib — FTP protocol client
  - 20.8.1. FTP Objects
  - 20.8.2. FTP_TLS Objects

- 20.9. poplib — POP3 protocol client
  - 20.9.1. POP3 Objects
  - 20.9.2. POP3 Example

- 20.10. imaplib — IMAP4 protocol client
  - 20.10.1. IMAP4 Objects
  - 20.10.2. IMAP4 Example

- 20.11. nntplib — NNTP protocol client
  - 20.11.1. NNTP Objects

- 20.12. smtplib — SMTP protocol client
20.12.1. SMTP Objects
20.12.2. SMTP Example

20.13. smtpd — SMTP Server
- 20.13.1. SMPTServer Objects
- 20.13.2. DebuggingServer Objects
- 20.13.3. PureProxy Objects
- 20.13.4. MailmanProxy Objects

20.14. telnetlib — Telnet client
- 20.14.1. Telnet Objects
- 20.14.2. Telnet Example

20.15. uuid — UUID objects according to RFC 4122
- 20.15.1. Example

20.16. urlparse — Parse URLs into components
- 20.16.1. Results of urlparse() and urlsplit()

20.17. SocketServer — A framework for network servers
- 20.17.1. Server Creation Notes
- 20.17.2. Server Objects
- 20.17.3. Request Handler Objects
- 20.17.4. Examples
  - 20.17.4.1. SocketServer.TCPServer Example
  - 20.17.4.2. SocketServer.UDPServer Example
  - 20.17.4.3. Asynchronous Mixins

20.18. BaseHTTPServer — Basic HTTP server
- 20.18.1. More examples

20.19. SimpleHTTPServer — Simple HTTP request handler

20.20. CGIHTTPServer — CGI-capable HTTP request handler

20.21. cookielib — Cookie handling for HTTP clients
- 20.21.2. FileCookieJar subclasses and cooperation with web browsers
- 20.21.3. CookiePolicy Objects
- 20.21.4. DefaultCookiePolicy Objects
- 20.21.5. Cookie Objects
- 20.21.6. Examples

- 20.22. **Cookie** — HTTP state management
  - 20.22.1. Cookie Objects
  - 20.22.2. Morsel Objects
  - 20.22.3. Example

- 20.23. **xmlrpclib** — XML-RPC client access
  - 20.23.1. ServerProxy Objects
  - 20.23.2. Boolean Objects
  - 20.23.3. DateTime Objects
  - 20.23.4. Binary Objects
  - 20.23.5. Fault Objects
  - 20.23.6. ProtocolError Objects
  - 20.23.7. MultiCall Objects
  - 20.23.8. Convenience Functions
  - 20.23.9. Example of Client Usage
  - 20.23.10. Example of Client and Server Usage

- 20.24. **SimpleXMLRPCServer** — Basic XML-RPC server
  - 20.24.1. SimpleXMLRPCServer Objects
    - 20.24.1.1. SimpleXMLRPCServer Example
  - 20.24.2. CGIXMLRPCRequestHandler

- 20.25. **DocXMLRPCServer** — Self-documenting XML-RPC server
  - 20.25.1. DocXMLRPCServer Objects
  - 20.25.2. DocCGIXMLRPCRequestHandler

- 21. Multimedia Services
  - 21.1. **audioop** — Manipulate raw audio data
  - 21.2. **imageop** — Manipulate raw image data
  - 21.3. **aifc** — Read and write AIFF and AIFC files
  - 21.4. **sunau** — Read and write Sun AU files
    - 21.4.1. AU_read Objects
- 21.4.2. AU_write Objects
- 21.5. wave — Read and write WAV files
  - 21.5.1. Wave_read Objects
  - 21.5.2. Wave_write Objects
- 21.6. chunk — Read IFF chunked data
- 21.7. colorsys — Conversions between color systems
- 21.8. imghdr — Determine the type of an image
- 21.9. sndhdr — Determine type of sound file
- 21.10. ossaudiodev — Access to OSS-compatible audio devices
  - 21.10.1. Audio Device Objects
  - 21.10.2. Mixer Device Objects

22. Internationalization
- 22.1. gettext — Multilingual internationalization services
  - 22.1.1. GNU gettext API
  - 22.1.2. Class-based API
    - 22.1.2.1. The NullTranslations class
    - 22.1.2.2. The GNUTranslations class
    - 22.1.2.3. Solaris message catalog support
    - 22.1.2.4. The Catalog constructor
  - 22.1.3. Internationalizing your programs and modules
    - 22.1.3.1. Localizing your module
    - 22.1.3.2. Localizing your application
    - 22.1.3.3. Changing languages on the fly
    - 22.1.3.4. Deferred translations
    - 22.1.3.5. gettext() vs. lgettext()
  - 22.1.4. Acknowledgements
- 22.2. locale — Internationalization services
  - 22.2.1. Background, details, hints, tips and caveats
  - 22.2.2. For extension writers and programs that
embed Python

- 22.2.3. Access to message catalogs

- 23. Program Frameworks
  - 23.1. cmd — Support for line-oriented command interpreters
    - 23.1.1. Cmd Objects
  - 23.2. shlex — Simple lexical analysis
    - 23.2.1. shlex Objects
    - 23.2.2. Parsing Rules

- 24. Graphical User Interfaces with Tk
  - 24.1. Tkinter — Python interface to Tcl/Tk
    - 24.1.1. Tkinter Modules
    - 24.1.2. Tkinter Life Preserver
      - 24.1.2.1. How To Use This Section
      - 24.1.2.2. A Simple Hello World Program
    - 24.1.3. A (Very) Quick Look at Tcl/Tk
    - 24.1.4. Mapping Basic Tk into Tkinter
    - 24.1.5. How Tk and Tkinter are Related
    - 24.1.6. Handy Reference
      - 24.1.6.1. Setting Options
      - 24.1.6.2. The Packer
      - 24.1.6.3. Packer Options
      - 24.1.6.4. Coupling Widget Variables
      - 24.1.6.5. The Window Manager
      - 24.1.6.6. Tk Option Data Types
      - 24.1.6.7. Bindings and Events
      - 24.1.6.8. The index Parameter
      - 24.1.6.9. Images
    - 24.1.7. File Handlers
  - 24.2. ttk — Tk themed widgets
    - 24.2.1. Using Ttk
    - 24.2.2. Ttk Widgets
    - 24.2.3. Widget
      - 24.2.3.1. Standard Options
- 24.2.3.2. Scrollable Widget Options
- 24.2.3.3. Label Options
- 24.2.3.4. Compatibility Options
- 24.2.3.5. Widget States
- 24.2.3.6. ttk.Widget

- 24.2.4. Combobox
  - 24.2.4.1. Options
  - 24.2.4.2. Virtual events
  - 24.2.4.3. ttk.Combobox

- 24.2.5. Notebook
  - 24.2.5.1. Options
  - 24.2.5.2. Tab Options
  - 24.2.5.3. Tab Identifiers
  - 24.2.5.4. Virtual Events
  - 24.2.5.5. ttk.Notebook

- 24.2.6. Progressbar
  - 24.2.6.1. Options
  - 24.2.6.2. ttk.Progressbar

- 24.2.7. Separator
  - 24.2.7.1. Options

- 24.2.8. Sizegrip
  - 24.2.8.1. Platform-specific notes
  - 24.2.8.2. Bugs

- 24.2.9. Treeview
  - 24.2.9.1. Options
  - 24.2.9.2. Item Options
  - 24.2.9.3. Tag Options
  - 24.2.9.4. Column Identifiers
  - 24.2.9.5. Virtual Events
  - 24.2.9.6. ttk.Treeview

- 24.2.10. Ttk Styling
  - 24.2.10.1. Layouts

- 24.3. Tix — Extension widgets for Tk
  - 24.3.1. Using Tix
24.3.2. Tix Widgets
   - 24.3.2.1. Basic Widgets
   - 24.3.2.2. File Selectors
   - 24.3.2.3. Hierarchical ListBox
   - 24.3.2.4. Tabular ListBox
   - 24.3.2.5. Manager Widgets
   - 24.3.2.6. Image Types
   - 24.3.2.7. Miscellaneous Widgets
   - 24.3.2.8. Form Geometry Manager

24.3.3. Tix Commands

24.4. ScrolledText — Scrolled Text Widget

24.5. turtle — Turtle graphics for Tk
   - 24.5.1. Introduction
   - 24.5.2. Overview over available Turtle and Screen methods
     - 24.5.2.1. Turtle methods
     - 24.5.2.2. Methods of TurtleScreen/Screen
   - 24.5.3. Methods of RawTurtle/Turtle and corresponding functions
     - 24.5.3.1. Turtle motion
     - 24.5.3.2. Tell Turtle’s state
     - 24.5.3.3. Settings for measurement
     - 24.5.3.4. Pen control
       - 24.5.3.4.1. Drawing state
       - 24.5.3.4.2. Color control
       - 24.5.3.4.3. Filling
       - 24.5.3.4.4. More drawing control
     - 24.5.3.5. Turtle state
       - 24.5.3.5.1. Visibility
       - 24.5.3.5.2. Appearance
     - 24.5.3.6. Using events
     - 24.5.3.7. Special Turtle methods
     - 24.5.3.8. Excursus about the use of compound shapes
24.5.4. Methods of TurtleScreen/Screen and corresponding functions
   - 24.5.4.1. Window control
   - 24.5.4.2. Animation control
   - 24.5.4.3. Using screen events
   - 24.5.4.4. Settings and special methods
   - 24.5.4.5. Methods specific to Screen, not inherited from TurtleScreen

24.5.5. The public classes of the module turtle

24.5.6. Help and configuration
   - 24.5.6.1. How to use help
   - 24.5.6.2. Translation of docstrings into different languages
   - 24.5.6.3. How to configure Screen and Turtles

24.5.7. Demo scripts

24.6. IDLE
   - 24.6.1. Menus
     - 24.6.1.1. File menu (Shell and Editor)
     - 24.6.1.2. Edit menu (Shell and Editor)
     - 24.6.1.3. Format menu (Editor window only)
     - 24.6.1.4. Run menu (Editor window only)
     - 24.6.1.5. Shell menu (Shell window only)
     - 24.6.1.6. Debug menu (Shell window only)
     - 24.6.1.7. Options menu (Shell and Editor)
     - 24.6.1.8. Window menu (Shell and Editor)
     - 24.6.1.9. Help menu (Shell and Editor)
     - 24.6.1.10. Context Menus
   - 24.6.2. Editing and navigation
     - 24.6.2.1. Automatic indentation
     - 24.6.2.2. Completions
     - 24.6.2.3. Calltips
     - 24.6.2.4. Python Shell window
     - 24.6.2.5. Text colors
   - 24.6.3. Startup and code execution
24.6.3.1. Command line usage
24.6.3.2. IDLE-console differences
24.6.3.3. Running without a subprocess
24.6.4. Help and preferences
   24.6.4.1. Additional help sources
   24.6.4.2. Setting preferences
   24.6.4.3. Extensions

24.7. Other Graphical User Interface Packages

25. Development Tools
   25.1. pydoc — Documentation generator and online help system
   25.2. doctest — Test interactive Python examples
      25.2.1. Simple Usage: Checking Examples in Docstrings
      25.2.2. Simple Usage: Checking Examples in a Text File
      25.2.3. How It Works
         25.2.3.1. Which Docstrings Are Examined?
         25.2.3.2. How are Docstring Examples Recognized?
         25.2.3.3. What’s the Execution Context?
         25.2.3.4. What About Exceptions?
         25.2.3.5. Option Flags
         25.2.3.6. Directives
         25.2.3.7. Warnings
      25.2.4. Basic API
      25.2.5. Unittest API
      25.2.6. Advanced API
         25.2.6.1. DocTest Objects
         25.2.6.2. Example Objects
         25.2.6.3. DocTestFinder objects
         25.2.6.4. DocTestParser objects
         25.2.6.5. DocTestRunner objects
         25.2.6.6. OutputChecker objects
- 25.2.7. Debugging
- 25.2.8. Soapbox
- 25.3. `unittest` — Unit testing framework
  - 25.3.1. Basic example
  - 25.3.2. Command-Line Interface
    - 25.3.2.1. Command-line options
  - 25.3.3. Test Discovery
  - 25.3.4. Organizing test code
  - 25.3.5. Re-using old test code
  - 25.3.6. Skipping tests and expected failures
  - 25.3.7. Classes and functions
    - 25.3.7.1. Test cases
      - 25.3.7.1.1. Deprecated aliases
    - 25.3.7.2. Grouping tests
    - 25.3.7.3. Loading and running tests
      - 25.3.7.3.1. load_tests Protocol
  - 25.3.8. Class and Module Fixtures
    - 25.3.8.1. setUpClass and tearDownClass
    - 25.3.8.2. setUpModule and tearDownModule
  - 25.3.9. Signal Handling
- 25.4. `2to3` - Automated Python 2 to 3 code translation
  - 25.4.1. Using 2to3
  - 25.4.2. Fixers
  - 25.4.3. `lib2to3` - 2to3’s library
- 25.5. `test` — Regression tests package for Python
  - 25.5.1. Writing Unit Tests for the test package
  - 25.5.2. Running tests using the command-line interface
- 25.6. `test.test_support` — Utility functions for tests
- 26. Debugging and Profiling
  - 26.1. `bdb` — Debugger framework
  - 26.2. `pdb` — The Python Debugger
  - 26.3. Debugger Commands
- 28.1. `sys` — System-specific parameters and functions
- 28.2. `sysconfig` — Provide access to Python's configuration information
  - 28.2.1. Configuration variables
  - 28.2.2. Installation paths
  - 28.2.3. Other functions
- 28.3. `__builtin__` — Built-in objects
- 28.4. `future_builtins` — Python 3 builtins
- 28.5. `__main__` — Top-level script environment
- 28.6. `warnings` — Warning control
  - 28.6.1. Warning Categories
  - 28.6.2. The Warnings Filter
    - 28.6.2.1. Default Warning Filters
  - 28.6.3. Temporarily Suppressing Warnings
  - 28.6.4. Testing Warnings
  - 28.6.5. Updating Code For New Versions of Python
  - 28.6.6. Available Functions
  - 28.6.7. Available Context Managers
- 28.7. `contextlib` — Utilities for `with`-statement contexts
- 28.8. `abc` — Abstract Base Classes
- 28.9. `atexit` — Exit handlers
  - 28.9.1. `atexit` Example
- 28.10. `traceback` — Print or retrieve a stack traceback
  - 28.10.1. Traceback Examples
- 28.11. `__future__` — Future statement definitions
- 28.12. `gc` — Garbage Collector interface
- 28.13. `inspect` — Inspect live objects
  - 28.13.1. Types and members
  - 28.13.2. Retrieving source code
  - 28.13.3. Classes and functions
  - 28.13.4. The interpreter stack
28.14. **site** — Site-specific configuration hook
28.15. **user** — User-specific configuration hook
28.16. **fpectl** — Floating point exception control
   - 28.16.1. Example
   - 28.16.2. Limitations and other considerations

- **29. Custom Python Interpreters**
  - 29.1. **code** — Interpreter base classes
    - 29.1.1. Interactive Interpreter Objects
    - 29.1.2. Interactive Console Objects
  - 29.2. **codeop** — Compile Python code

- **30. Restricted Execution**
  - 30.1. **reexec** — Restricted execution framework
    - 30.1.1. RExec Objects
    - 30.1.2. Defining restricted environments
    - 30.1.3. An example
  - 30.2. **Bastion** — Restricting access to objects

- **31. Importing Modules**
  - 31.1. **imp** — Access the `import` internals
    - 31.1.1. Examples
  - 31.2. **importlib** — Convenience wrappers for `__import__()`
  - 31.3. **imputil** — Import utilities
    - 31.3.1. Examples
  - 31.4. **zipimport** — Import modules from Zip archives
    - 31.4.1. zipimporter Objects
    - 31.4.2. Examples
  - 31.5. **pkgutil** — Package extension utility
  - 31.6. **modulefinder** — Find modules used by a script
    - 31.6.1. Example usage of `ModuleFinder`
  - 31.7. **runpy** — Locating and executing Python modules

- **32. Python Language Services**
  - 32.1. **parser** — Access Python parse trees
32.1. Creating ST Objects
32.1.2. Converting ST Objects
32.1.3. Queries on ST Objects
32.1.4. Exceptions and Error Handling
32.1.5. ST Objects
32.1.6. Example: Emulation of \texttt{compile()}

32.2. \texttt{ast} — Abstract Syntax Trees
   32.2.1. Node classes
   32.2.2. Abstract Grammar
   32.2.3. \texttt{ast} Helpers

32.3. \texttt{symtable} — Access to the compiler’s symbol tables
   32.3.1. Generating Symbol Tables
   32.3.2. Examining Symbol Tables

32.4. \texttt{symbol} — Constants used with Python parse trees
32.5. \texttt{token} — Constants used with Python parse trees
32.6. \texttt{keyword} — Testing for Python keywords
32.7. \texttt{tokenize} — Tokenizer for Python source
32.8. \texttt{tabnanny} — Detection of ambiguous indentation
32.9. \texttt{pyclbr} — Python class browser support
   32.9.1. Class Objects
   32.9.2. Function Objects
32.10. \texttt{py_compile} — Compile Python source files
32.11. \texttt{compileall} — Byte-compile Python libraries
   32.11.1. Command-line use
   32.11.2. Public functions
32.12. \texttt{dis} — Disassembler for Python bytecode
   32.12.1. Python Bytecode Instructions
32.13. \texttt{pickletools} — Tools for pickle developers

33. Python compiler package
   33.1. The basic interface
   33.2. Limitations
33.3. Python Abstract Syntax
   - 33.3.1. AST Nodes
   - 33.3.2. Assignment nodes
   - 33.3.3. Examples

33.4. Using Visitors to Walk ASTs

33.5. Bytecode Generation

34. Miscellaneous Services
   - 34.1. formatter — Generic output formatting
     - 34.1.1. The Formatter Interface
     - 34.1.2. Formatter Implementations
     - 34.1.3. The Writer Interface
     - 34.1.4. Writer Implementations

35. MS Windows Specific Services
   - 35.1. msilib — Read and write Microsoft Installer files
     - 35.1.1. Database Objects
     - 35.1.2. View Objects
     - 35.1.3. Summary Information Objects
     - 35.1.4. Record Objects
     - 35.1.5. Errors
     - 35.1.6. CAB Objects
     - 35.1.7. Directory Objects
     - 35.1.8. Features
     - 35.1.9. GUI classes
     - 35.1.10. Precomputed tables

   - 35.2. msvcrt – Useful routines from the MS VC++ runtime
     - 35.2.1. File Operations
     - 35.2.2. Console I/O
     - 35.2.3. Other Functions

   - 35.3. _winreg – Windows registry access
     - 35.3.1. Constants
       - 35.3.1.1. HKEY_* Constants
       - 35.3.1.2. Access Rights
         - 35.3.1.2.1. 64-bit Specific
35.3.1.3. Value Types
35.3.2. Registry Handle Objects
35.4. **winsound** — Sound-playing interface for Windows

36. **Unix Specific Services**
   - 36.1. **posix** — The most common POSIX system calls
     - 36.1.1. Large File Support
     - 36.1.2. Notable Module Contents
   - 36.2. **pwd** — The password database
   - 36.3. **spwd** — The shadow password database
   - 36.4. **grp** — The group database
   - 36.5. **crypt** — Function to check Unix passwords
   - 36.6. **dl** — Call C functions in shared objects
     - 36.6.1. DI Objects
   - 36.7. **termios** — POSIX style tty control
     - 36.7.1. Example
   - 36.8. **tty** — Terminal control functions
   - 36.9. **pty** — Pseudo-terminal utilities
   - 36.10. **fcntl** — The `fcntl` and `ioctl` system calls
   - 36.11. **pipes** — Interface to shell pipelines
     - 36.11.1. Template Objects
   - 36.12. **posixfile** — File-like objects with locking support
   - 36.13. **resource** — Resource usage information
     - 36.13.1. Resource Limits
     - 36.13.2. Resource Usage
   - 36.14. **nis** — Interface to Sun’s NIS (Yellow Pages)
   - 36.15. **syslog** — Unix syslog library routines
     - 36.15.1. Examples
       - 36.15.1.1. Simple example
   - 36.16. **commands** — Utilities for running commands

37. **Mac OS X specific services**
   - 37.1. **ic** — Access to the Mac OS X Internet Config
37.1.1. IC Objects

37.2. MacOS — Access to Mac OS interpreter features

37.3. macostools — Convenience routines for file manipulation

37.4. findertools — The finder's Apple Events interface

37.5. EasyDialogs — Basic Macintosh dialogs
  - 37.5.1. ProgressBar Objects

37.6. FrameWork — Interactive application framework
  - 37.6.1. Application Objects
  - 37.6.2. Window Objects
  - 37.6.3. ControlsWindow Object
  - 37.6.4. ScrolledWindow Object
  - 37.6.5. DialogWindow Objects

37.7. autoGIL — Global Interpreter Lock handling in event loops

37.8. Mac OS Toolbox Modules
  - 37.8.1. Carbon.AE — Apple Events
  - 37.8.2. Carbon.AH — Apple Help
  - 37.8.3. Carbon.App — Appearance Manager
  - 37.8.4. Carbon.Appearance — Appearance Manager constants
  - 37.8.5. Carbon.CF — Core Foundation
  - 37.8.6. Carbon.CG — Core Graphics
  - 37.8.7. Carbon.CarbonEvt — Carbon Event Manager
  - 37.8.9. Carbon.Cm — Component Manager
  - 37.8.10. Carbon.Components — Component Manager constants
  - 37.8.11. Carbon.ControlAccessor — Control
Manager accessor

- 37.8.12. `Carbon.Controls` — Control Manager constants
- 37.8.13. `Carbon.CoreFoundation` — CoreFoundation constants
- 37.8.15. `Carbon.Ctl` — Control Manager
- 37.8.16. `Carbon.Dials` — Dialog Manager constants
- 37.8.17. `Carbon.Dlg` — Dialog Manager
- 37.8.18. `Carbon.Drag` — Drag and Drop Manager
- 37.8.19. `Carbon.Dragconst` — Drag and Drop Manager constants
- 37.8.20. `Carbon.Events` — Event Manager constants
- 37.8.22. `Carbon.File` — File Manager
- 37.8.23. `Carbon.Files` — File Manager constants
- 37.8.24. `Carbon.Fm` — Font Manager
- 37.8.25. `Carbon.Folder` — Folder Manager
- 37.8.27. `Carbon.Fonts` — Font Manager constants
- 37.8.28. `Carbon.Help` — Help Manager
- 37.8.29. `Carbon.IBCarbon` — Carbon InterfaceBuilder
- 37.8.30. `Carbon.IBCarbonRuntime` — Carbon InterfaceBuilder constants
- 37.8.31. `Carbon.Icn` — Carbon Icon Manager
- 37.8.32. `Carbon.Icons` — Carbon Icon Manager constants
37.8.33. **Carbon.Launch** — Carbon Launch Services
37.8.34. **Carbon.LaunchServices** — Carbon Launch Services constants
37.8.35. **Carbon.List** — List Manager
37.8.36. **Carbon.Lists** — List Manager constants
37.8.37. **Carbon.MacHelp** — Help Manager constants
37.8.38. **Carbon.MediaDescr** — Parsers and generators for Quicktime Media descriptors
37.8.39. **Carbon.Menu** — Menu Manager
37.8.40. **Carbon.Menus** — Menu Manager constants
37.8.41. **Carbon.Mlte** — MultiLingual Text Editor
37.8.42. **Carbon.OSA** — Carbon OSA Interface
37.8.43. **Carbon.OSAconst** — Carbon OSA Interface constants
37.8.44. **Carbon.QDOffscreen** — QuickDraw Offscreen constants
37.8.45. **Carbon.Qd** — QuickDraw
37.8.46. **Carbon.Qdoffs** — QuickDraw Offscreen
37.8.47. **Carbon.Qt** — QuickTime
37.8.48. **Carbon.QuickDraw** — QuickDraw constants
37.8.49. **Carbon.QuickTime** — QuickTime constants
37.8.50. **Carbon.Res** — Resource Manager and Handles
37.8.51. **Carbon.Resources** — Resource Manager and Handles constants
37.8.52. **Carbon.Scrap** — Scrap Manager
37.8.53. **Carbon.Snd** — Sound Manager
- 37.8.54. **Carbon.Sound** — Sound Manager constants
- 37.8.55. **Carbon.TE** — TextEdit
- 37.8.56. **Carbon.TextEdit** — TextEdit constants
- 37.8.57. **Carbon.Win** — Window Manager
- 37.8.58. **Carbon.Windows** — Window Manager constants
- 37.9. **ColorPicker** — Color selection dialog

38. MacPython OSA Modules
- 38.1. **gensuitemodule** — Generate OSA stub packages
- 38.2. **aetools** — OSA client support
- 38.3. **aepack** — Conversion between Python variables and AppleEvent data containers
- 38.4. **aetypes** — AppleEvent objects
- 38.5. **MiniAEFrame** — Open Scripting Architecture server support
  - 38.5.1. AEServer Objects

39. SGI IRIX Specific Services
- 39.1. **a1** — Audio functions on the SGI
  - 39.1.1. Configuration Objects
  - 39.1.2. Port Objects
- 39.2. **AL** — Constants used with the **a1** module
- 39.3. **cd** — CD-ROM access on SGI systems
  - 39.3.1. Player Objects
  - 39.3.2. Parser Objects
- 39.4. **f1** — FORMS library for graphical user interfaces
  - 39.4.1. Functions Defined in Module **f1**
  - 39.4.2. Form Objects
  - 39.4.3. FORMS Objects
- 39.5. **FL** — Constants used with the **f1** module
- 39.6. **f1p** — Functions for loading stored FORMS designs
39.7. fm — *Font Manager* interface
39.8. gl — *Graphics Library* interface
39.9. DEVICE — Constants used with the *gl* module
39.10. GL — Constants used with the *gl* module
39.11. imgfile — Support for SGI imglib files
39.12. jpeg — Read and write JPEG files

- 40. SunOS Specific Services
  - 40.1. sunaudiodev — Access to Sun audio hardware
    - 40.1.1. Audio Device Objects
  - 40.2. SUNAUDIODEV — Constants used with *sunaudiodev*

- 41. Undocumented Modules
  - 41.1. Miscellaneous useful utilities
  - 41.2. Platform specific modules
  - 41.3. Multimedia
  - 41.4. Undocumented Mac OS modules
    - 41.4.1. applesingle — Apple Single decoder
    - 41.4.2. buildtools — Helper module for BuildApplet and Friends
    - 41.4.3. cfmfile — Code Fragment Resource module
    - 41.4.4. icopen — Internet Config replacement for open()
    - 41.4.5. macerrors — Mac OS Errors
    - 41.4.6. macresource — Locate script resources
    - 41.4.7. Nav — NavServices calls
    - 41.4.8. PixMapWrapper — Wrapper for PixMap objects
    - 41.4.9. videoreader — Read QuickTime movies
    - 41.4.10. W — Widgets built on FrameWork

- 41.5. Obsolete
- 41.6. SGI-specific Extension modules

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

2. Defining New Types
   2.1. The Basics
      2.1.1. Adding data and methods to the Basic example
      2.1.2. Providing finer control over data attributes
      2.1.3. Supporting cyclic garbage collection
      2.1.4. Subclassing other types
   2.2. Type Methods
      2.2.1. Finalization and De-allocation
      2.2.2. Object Presentation
      2.2.3. Attribute Management
         2.2.3.1. Generic Attribute Management
         2.2.3.2. Type-specific Attribute Management
      2.2.4. Object Comparison
      2.2.5. Abstract Protocol Support
      2.2.6. Weak Reference Support
- 2.2.7. More Suggestions
- 3. Building C and C++ Extensions with distutils
  - 3.1. Distributing your extension modules
- 4. Building C and C++ Extensions on Windows
  - 4.1. A Cookbook Approach
  - 4.2. Differences Between Unix and Windows
  - 4.3. Using DLLs in Practice
- 5. Embedding Python in Another Application
  - 5.1. Very High Level Embedding
  - 5.2. Beyond Very High Level Embedding: An overview
  - 5.3. Pure Embedding
  - 5.4. Extending Embedded Python
  - 5.5. Embedding Python in C++
  - 5.6. Compiling and Linking under Unix-like systems

- Python/C API Reference Manual
  - Introduction
    - Include Files
    - Objects, Types and Reference Counts
      - Reference Counts
        - Reference Count Details
      - Types
    - Exceptions
    - Embedding Python
    - Debugging Builds
  - The Very High Level Layer
  - Reference Counting
  - Exception Handling
    - Unicode Exception Objects
    - Recursion Control
    - Standard Exceptions
    - String Exceptions
  - Utilities
    - Operating System Utilities
    - System Functions
- Process Control
- Importing Modules
- Data marshalling support
- Parsing arguments and building values
- String conversion and formatting
- Reflection
- Codec registry and support functions
  - Codec lookup API
  - Registry API for Unicode encoding error handlers
- Abstract Objects Layer
  - Object Protocol
  - Number Protocol
  - Sequence Protocol
  - Mapping Protocol
  - Iterator Protocol
  - Old Buffer Protocol
- Concrete Objects Layer
  - Fundamental Objects
    - Type Objects
    - The `None` Object
  - Numeric Objects
    - Plain Integer Objects
    - Boolean Objects
    - Long Integer Objects
    - Floating Point Objects
    - Complex Number Objects
      - Complex Numbers as C Structures
      - Complex Numbers as Python Objects
  - Sequence Objects
    - Byte Array Objects
      - Type check macros
      - Direct API functions
      - Macros
    - String/Bytes Objects
- Unicode Objects and Codecs
  - Unicode Objects
    - Unicode Type
    - Unicode Character Properties
    - Plain Py_UNICODE
    - wchar_t Support
  - Built-in Codecs
    - Generic Codecs
    - UTF-8 Codecs
    - UTF-32 Codecs
    - UTF-16 Codecs
    - UTF-7 Codecs
    - Unicode-Escape Codecs
    - Raw-Unicode-Escape Codecs
    - Latin-1 Codecs
    - ASCII Codecs
    - Character Map Codecs
    - MBCS codecs for Windows
    - Methods & Slots
      - Methods and Slot Functions
  - Buffers and Memoryview Objects
    - The new-style Py_buffer struct
    - Buffer related functions
    - MemoryView objects
    - Old-style buffer objects
  - Tuple Objects
  - List Objects
- Mapping Objects
  - Dictionary Objects
- Other Objects
  - Class and Instance Objects
  - Function Objects
  - Method Objects
  - File Objects
- Module Objects
- Iterator Objects
- Descriptor Objects
- Slice Objects
- Weak Reference Objects
- Capsules
- CObjects
- Cell Objects
- Generator Objects
- DateTime Objects
- Set Objects
- Code Objects

- Initialization, Finalization, and Threads
  - Initializing and finalizing the interpreter
  - Process-wide parameters
  - Thread State and the Global Interpreter Lock
    - Releasing the GIL from extension code
    - Non-Python created threads
  - High-level API
  - Low-level API
  - Sub-interpreter support
    - Bugs and caveats
- Asynchronous Notifications
- Profiling and Tracing
- Advanced Debugger Support

- Memory Management
  - Overview
  - Memory Interface
  - Examples

- Object Implementation Support
  - Allocating Objects on the Heap
  - Common Object Structures
  - Type Objects
  - Number Object Structures
- Mapping Object Structures
- Sequence Object Structures
- Buffer Object Structures
- Supporting Cyclic Garbage Collection

- Distributing Python Modules
  - Key terms
  - Open source licensing and collaboration
  - Installing the tools
  - Reading the guide
  - How do I...?
    - ... choose a name for my project?
    - ... create and distribute binary extensions?

- Installing Python Modules
  - Key terms
  - Basic usage
  - How do I ...?
    - ... install pip in versions of Python prior to Python 2.7.9?
    - ... install packages just for the current user?
    - ... install scientific Python packages?
    - ... work with multiple versions of Python installed in parallel?
  - Common installation issues
    - Installing into the system Python on Linux
    - Installing binary extensions

- Python HOWTOs
  - Porting Python 2 Code to Python 3
    - The Short Explanation
    - Details
      - Drop support for Python 2.5 and older (at least)
      - Make sure you specify the proper version support in your setup.py file
      - Have good test coverage
      - Learn the differences between Python 2 & 3
- Update your code
  - Division
  - Text versus binary data
- Prevent compatibility regressions
- Check which dependencies block your transition
- Update your setup.py file to denote Python 3 compatibility
- Use continuous integration to stay compatible
- Dropping Python 2 support completely
  - Porting Extension Modules to Python 3
    - Conditional compilation
    - Changes to Object APIs
      - str/unicode Unification
      - long/int Unification
    - Module initialization and state
    - CObject replaced with Capsule
    - Other options
  - Curses Programming with Python
    - What is curses?
      - The Python curses module
    - Starting and ending a curses application
    - Windows and Pads
    - Displaying Text
      - Attributes and Color
    - User Input
    - For More Information
  - Descriptor HowTo Guide
    - Abstract
    - Definition and Introduction
    - Descriptor Protocol
    - Invoking Descriptors
    - Descriptor Example
    - Properties
    - Functions and Methods
- Static Methods and Class Methods
- Idioms and Anti-Idioms in Python
  - Language Constructs You Should Not Use
    - from module import *
      - Inside Function Definitions
      - At Module Level
      - When It Is Just Fine
    - Unadorned `exec`, `execfile()` and friends
    - from module import name1, name2
    - except:
  - Exceptions
  - Using the Batteries
  - Using Backslash to Continue Statements
- Functional Programming HOWTO
  - Introduction
    - Formal provability
    - Modularity
    - Ease of debugging and testing
    - Composability
  - Iterators
    - Data Types That Support Iterators
  - Generator expressions and list comprehensions
  - Generators
    - Passing values into a generator
  - Built-in functions
  - Small functions and the lambda expression
  - The `itertools` module
    - Creating new iterators
    - Calling functions on elements
    - Selecting elements
    - Grouping elements
  - The `functools` module
    - The operator module
  - Revision History and Acknowledgements
- References
  - General
  - Python-specific
  - Python documentation
- Logging HOWTO
  - Basic Logging Tutorial
    - When to use logging
    - A simple example
    - Logging to a file
    - Logging from multiple modules
    - Logging variable data
    - Changing the format of displayed messages
    - Displaying the date/time in messages
    - Next Steps
  - Advanced Logging Tutorial
    - Logging Flow
    - Loggers
    - Handlers
    - Formatters
    - Configuring Logging
      - What happens if no configuration is provided
      - Configuring Logging for a Library
    - Logging Levels
      - Custom Levels
    - Useful Handlers
    - Exceptions raised during logging
    - Using arbitrary objects as messages
    - Optimization
- Logging Cookbook
  - Using logging in multiple modules
  - Logging from multiple threads
  - Multiple handlers and formatters
  - Logging to multiple destinations
  - Configuration server example
- Sending and receiving logging events across a network
- Adding contextual information to your logging output
  - Using LoggerAdapters to impart contextual information
  - Using objects other than dicts to pass contextual information
  - Using Filters to impart contextual information
- Logging to a single file from multiple processes
- Using file rotation
- An example dictionary-based configuration
- Inserting a BOM into messages sent to a SysLogHandler
- Implementing structured logging
- Customizing handlers with \texttt{dictConfig()}
- Configuring filters with \texttt{dictConfig()}
- Customized exception formatting
- Speaking logging messages
- Buffering logging messages and outputting them conditionally
- Formatting times using UTC (GMT) via configuration
- Using a context manager for selective logging

- Regular Expression HOWTO
  - Introduction
  - Simple Patterns
    - Matching Characters
    - Repeating Things
  - Using Regular Expressions
    - Compiling Regular Expressions
    - The Backslash Plague
    - Performing Matches
    - Module-Level Functions
    - Compilation Flags
  - More Pattern Power
    - More Metacharacters
- Grouping
  - Non-capturing and Named Groups
  - Lookahead Assertions
- Modifying Strings
  - Splitting Strings
  - Search and Replace
- Common Problems
  - Use String Methods
  - match() versus search()
  - Greedy versus Non-Greedy
  - Using re.VERBOSE
- Feedback
  - Socket Programming HOWTO
    - Sockets
      - History
    - Creating a Socket
      - IPC
    - Using a Socket
      - Binary Data
    - Disconnecting
      - When Sockets Die
    - Non-blocking Sockets
      - Performance
  - Sorting HOW TO
    - Sorting Basics
    - Key Functions
    - Operator Module Functions
    - Ascending and Descending
    - Sort Stability and Complex Sorts
    - The Old Way Using Decorate-Sort-Undecorate
    - The Old Way Using the cmp Parameter
    - Odd and Ends
  - Unicode HOWTO
    - Introduction to Unicode
- History of Character Codes
- Definitions
- Encodings
- References
- Python 2.x’s Unicode Support
  - The Unicode Type
  - Unicode Literals in Python Source Code
  - Unicode Properties
  - References
- Reading and Writing Unicode Data
  - Unicode filenames
  - Tips for Writing Unicode-aware Programs
  - References
- Revision History and Acknowledgements
  - HOWTO Fetch Internet Resources Using urllib2
    - Introduction
    - Fetching URLs
      - Data
      - Headers
    - Handling Exceptions
      - URLError
      - HTTPError
        - Error Codes
    - Wrapping it Up
      - Number 1
      - Number 2
  - info and geturl
  - Openers and Handlers
  - Basic Authentication
  - Proxies
  - Sockets and Layers
  - Footnotes
- HOWTO Use Python in the web
  - The Low-Level View
- Common Gateway Interface
  - Simple script for testing CGI
  - Setting up CGI on your own server
  - Common problems with CGI scripts
- mod_python
- FastCGI and SCGI
  - Setting up FastCGI
- mod_wsgi
- Step back: WSGI
  - WSGI Servers
  - Case study: MoinMoin
- Model-View-Controller
- Ingredients for Websites
  - Templates
  - Data persistence
- Frameworks
  - Some notable frameworks
    - Django
    - TurboGears
    - Zope
    - Other notable frameworks
- Argparse Tutorial
  - Concepts
  - The basics
  - Introducing Positional arguments
  - Introducing Optional arguments
    - Short options
  - Combining Positional and Optional arguments
  - Getting a little more advanced
    - Conflicting options
  - Conclusion
- Python Frequently Asked Questions
  - General Python FAQ
    - General Information
- Python in the real world
- Upgrading Python

**Programming FAQ**
- General Questions
- Core Language
- Numbers and strings
- Sequences (Tuples/Lists)
- Dictionaries
- Objects
- Modules

**Design and History FAQ**
- Why does Python use indentation for grouping of statements?
- Why am I getting strange results with simple arithmetic operations?
- Why are floating point calculations so inaccurate?
- Why are Python strings immutable?
- Why must ‘self’ be used explicitly in method definitions and calls?
- Why can’t I use an assignment in an expression?
- Why does Python use methods for some functionality (e.g. list.index()) but functions for other (e.g. len(list))?
- Why is join() a string method instead of a list or tuple method?
- How fast are exceptions?
- Why isn’t there a switch or case statement in Python?
- Can’t you emulate threads in the interpreter instead of relying on an OS-specific thread implementation?
- Why can’t lambda expressions contain statements?
- Can Python be compiled to machine code, C or some other language?
- How does Python manage memory?
- Why isn’t all memory freed when Python exits?
- Why are there separate tuple and list data types?
- How are lists implemented?
- How are dictionaries implemented?
- Why must dictionary keys be immutable?
- Why doesn't list.sort() return the sorted list?
- How do you specify and enforce an interface spec in Python?
- Why is there no goto?
- Why can't raw strings (r-strings) end with a backslash?
- Why doesn't Python have a "with" statement for attribute assignments?
- Why are colons required for the if/while/def/class statements?
- Why does Python allow commas at the end of lists and tuples?

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

- Extending/Embedding FAQ
  - Can I create my own functions in C?
  - Can I create my own functions in C++?
  - Writing C is hard; are there any alternatives?
  - How can I execute arbitrary Python statements from C?
  - How can I evaluate an arbitrary Python expression from C?
  - How do I extract C values from a Python object?
  - How do I use Py_BuildValue() to create a tuple of arbitrary length?
  - How do I call an object's method from C?
- How do I catch the output from PyErr_Print() (or anything that prints to stdout/stderr)?
- How do I access a module written in Python from C?
- How do I interface to C++ objects from Python?
- I added a module using the Setup file and the make fails; why?
- How do I debug an extension?
- I want to compile a Python module on my Linux system, but some files are missing. Why?
- What does “SystemError: _PyImport_FixupExtension: module yourmodule not loaded” mean?
- How do I tell “incomplete input” from “invalid input”?
- How do I find undefined g++ symbols __builtin_new or __pure_virtual?
- Can I create an object class with some methods implemented in C and others in Python (e.g. through inheritance)?
- When importing module X, why do I get “undefined symbol: PyUnicodeUCS2”??

**Python on Windows FAQ**
- How do I run a Python program under Windows?
- How do I make Python scripts executable?
- Why does Python sometimes take so long to start?
- How do I make an executable from a Python script?
- Is a *.pyd file the same as a DLL?
- How can I embed Python into a Windows application?
- How do I keep editors from inserting tabs into my Python source?
- How do I check for a keypress without blocking?
- How do I emulate os.kill() in Windows?
- How do I extract the downloaded documentation on Windows?

**Graphic User Interface FAQ**
- What platform-independent GUI toolkits exist for
Python?
- What platform-specific GUI toolkits exist for Python?
- Tkinter questions
  - “Why is Python Installed on my Computer?” FAQ
    - What is Python?
    - Why is Python installed on my machine?
    - Can I delete Python?

- Glossary
- About these documents
  - Contributors to the Python Documentation
- Reporting Bugs
  - Documentation bugs
  - Using the Python issue tracker
  - Getting started contributing to Python yourself
- Copyright
- History and License
  - History of the software
  - Terms and conditions for accessing or otherwise using Python
    - PSF LICENSE AGREEMENT FOR PYTHON 2.7.13
    - BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0
    - CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1
    - CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2
  - Licenses and Acknowledgements for Incorporated Software
    - Mersenne Twister
    - Sockets
    - Floating point exception control
    - MD5 message digest algorithm
    - Asynchronous socket services
    - Cookie management
    - Execution tracing
    - UUencode and UUdecode functions
- XML Remote Procedure Calls
- test_epoll
- Select kqueue
- strtod and dtoa
- OpenSSL
- expat
- libffi
- zlib
10. API Reference
10.1. distutils.core — Core Distutils functionality

The `distutils.core` module is the only module that needs to be installed to use the Distutils. It provides the `setup()` (which is called from the setup script). Indirectly provides the `distutils.dist.Distribution` and `distutils.cmd.Command` class.

**distutils.core.** `setup(arguments)`

The basic do-everything function that does most everything you could ever ask for from a Distutils method.

The setup function takes a large number of arguments. These are laid out in the following table.

<table>
<thead>
<tr>
<th>argument name</th>
<th>value</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>name</code></td>
<td>The name of the package</td>
<td>a string</td>
</tr>
<tr>
<td><code>version</code></td>
<td>The version number of the package; see <code>distutils.version</code></td>
<td>a string</td>
</tr>
<tr>
<td><code>description</code></td>
<td>A single line describing the package</td>
<td>a string</td>
</tr>
<tr>
<td><code>long_description</code></td>
<td>Longer description of the package</td>
<td>a string</td>
</tr>
<tr>
<td><code>author</code></td>
<td>The name of the package author</td>
<td>a string</td>
</tr>
<tr>
<td><code>author_email</code></td>
<td>The email address of the package author</td>
<td>a string</td>
</tr>
<tr>
<td></td>
<td>The name of the current maintainer,</td>
<td></td>
</tr>
<tr>
<td><strong>maintainer</strong></td>
<td>if different from the author. Note that if the maintainer is provided, distutils will use it as the author in <strong>PKG-INFO</strong></td>
<td>a string</td>
</tr>
<tr>
<td><strong>maintainer_email</strong></td>
<td>The email address of the current maintainer, if different from the author</td>
<td>a string</td>
</tr>
<tr>
<td><strong>url</strong></td>
<td>A URL for the package (homepage)</td>
<td>a string</td>
</tr>
<tr>
<td><strong>download_url</strong></td>
<td>A URL to download the package</td>
<td>a string</td>
</tr>
<tr>
<td><strong>packages</strong></td>
<td>A list of Python packages that distutils will manipulate</td>
<td>a list of strings</td>
</tr>
<tr>
<td><strong>py_modules</strong></td>
<td>A list of Python modules that distutils will manipulate</td>
<td>a list of strings</td>
</tr>
<tr>
<td><strong>scripts</strong></td>
<td>A list of standalone script files to be built and installed</td>
<td>a list of strings</td>
</tr>
<tr>
<td><strong>ext_modules</strong></td>
<td>A list of Python extensions to be built</td>
<td>a list of instances of <code>distutils.core.Extension</code></td>
</tr>
<tr>
<td><strong>classifiers</strong></td>
<td>A list of categories for the package</td>
<td>a list of strings; valid classifiers are listed on PyPI.</td>
</tr>
<tr>
<td><strong>distclass</strong></td>
<td>the <code>Distribution</code> class to use</td>
<td>a subclass of <code>distutils.core.Distribution</code></td>
</tr>
</tbody>
</table>
The name of the setup.py script - defaults to sys.argv[0]

Arguments to supply to the setup script

default options for the setup script

The license for the package

Descriptive metadata, see PEP 314

A mapping of command names to Command subclasses

A list of data files to install

A mapping of package to directory names

The name of the setup.py script - defaults to sys.argv[0]

Arguments to supply to the setup script

default options for the setup script

The license for the package

Descriptive metadata, see PEP 314

A mapping of command names to Command subclasses

A list of data files to install

A mapping of package to directory names

Run a setup script in a somewhat controlled environment, and return the distutils.dist.Distribution instance that drives things. This is useful if you need to find out the distribution metadata (passed as keyword args from script to setup()), or the contents of the config files or command-line.

script_name is a file that will be run with execfile() sys.argv[0]
will be replaced with \textit{script} for the duration of the call. \textit{script\_args} is a list of strings; if supplied, \texttt{sys.argv[1:]} will be replaced by \textit{script\_args} for the duration of the call.

\textit{stop\_after} tells \texttt{setup()} when to stop processing; possible values:

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{init}</td>
<td>Stop after the \texttt{Distribution} instance has been created and populated with the keyword arguments to \texttt{setup()}</td>
</tr>
<tr>
<td>\textit{config}</td>
<td>Stop after config files have been parsed (and their data stored in the \texttt{Distribution} instance)</td>
</tr>
<tr>
<td>\textit{commandline}</td>
<td>Stop after the command-line (\texttt{sys.argv[1:]} or \textit{script_args}) have been parsed (and the data stored in the \texttt{Distribution} instance.)</td>
</tr>
<tr>
<td>\textit{run}</td>
<td>Stop after all commands have been run (the same as if \texttt{setup()} had been called in the usual way). This is the default value.</td>
</tr>
</tbody>
</table>

In addition, the \texttt{distutils.core} module exposed a number of classes that live elsewhere.

- \texttt{Extension} from \texttt{distutils.extension}
- \texttt{Command} from \texttt{distutils.cmd}
- \texttt{Distribution} from \texttt{distutils.dist}

A short description of each of these follows, but see the relevant module for the full reference.

\texttt{class} \texttt{distutils.core.Extension}

The Extension class describes a single C or C++ extension module in a setup script. It accepts the following keyword arguments in its constructor

<table>
<thead>
<tr>
<th>argument name</th>
<th>value</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name</strong></td>
<td>the full name of the extension, including any packages — <em>not</em> a filename or pathname, but Python dotted name</td>
<td>a string</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td><strong>sources</strong></td>
<td>list of source filenames, relative to the distribution root (where the setup script lives), in Unix form (slash-separated) for portability. Source files may be C, C++, SWIG (.i), platform-specific resource files, or whatever else is recognized by the <code>build_ext</code> command as source for a Python extension.</td>
<td>a list of strings</td>
</tr>
<tr>
<td><strong>include_dirs</strong></td>
<td>list of directories to search for C/C++ header files (in Unix form for portability)</td>
<td>a list of strings</td>
</tr>
<tr>
<td><strong>define_macros</strong></td>
<td>list of macros to define; each macro is defined using a 2-tuple <code>(name, value)</code>, where <code>value</code> is either the string to define it to or <code>None</code> to define it without a particular value (equivalent of <code>#define FOO</code> in source or <code>-DFOO</code> on Unix C compiler command line)</td>
<td>a list of tuples</td>
</tr>
<tr>
<td><strong>undef_macros</strong></td>
<td>list of macros to</td>
<td>a list of strings</td>
</tr>
<tr>
<td><strong>variable</strong></td>
<td><strong>description</strong></td>
<td><strong>type</strong></td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td><code>library_dirs</code></td>
<td>list of directories to search for C/C++ libraries at link time</td>
<td>a list of strings</td>
</tr>
<tr>
<td><code>libraries</code></td>
<td>list of library names (not filenames or paths) to link against</td>
<td>a list of strings</td>
</tr>
<tr>
<td><code>runtime_library_dirs</code></td>
<td>list of directories to search for C/C++ libraries at run time (for shared extensions, this is when the extension is loaded)</td>
<td>a list of strings</td>
</tr>
<tr>
<td><code>extra_objects</code></td>
<td>list of extra files to link with (eg. object files not implied by ‘sources’, static library that must be explicitly specified, binary resource files, etc.)</td>
<td>a list of strings</td>
</tr>
<tr>
<td><code>extra_compile_args</code></td>
<td>any extra platform- and compiler-specific information to use when compiling the source files in ‘sources’. For platforms and compilers where a command line makes sense, this is typically a list of command-line arguments, but for other platforms it could be anything.</td>
<td>a list of strings</td>
</tr>
<tr>
<td><code>extra_link_args</code></td>
<td>any extra platform- and compiler-specific information to use when linking object files</td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
<td>Type</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td><code>extra_link_args</code></td>
<td>together to create the extension (or to create a new static Python interpreter). Similar interpretation as for ‘extra_compile_args’.</td>
<td>a list of strings</td>
</tr>
<tr>
<td><code>export_symbols</code></td>
<td>list of symbols to be exported from a shared extension. Not used on all platforms, and not generally necessary for Python extensions, which typically export exactly one symbol: init + extension_name.</td>
<td>a list of strings</td>
</tr>
<tr>
<td><code>depends</code></td>
<td>list of files that the extension depends on</td>
<td>a list of strings</td>
</tr>
<tr>
<td><code>language</code></td>
<td>extension language (i.e. 'c', 'c++', 'objc'). Will be detected from the source extensions if not provided.</td>
<td>a string</td>
</tr>
</tbody>
</table>

**class distutils.core.Distribution**

A `Distribution` describes how to build, install and package up a Python software package.

See the `setup()` function for a list of keyword arguments accepted by the Distribution constructor. `setup()` creates a Distribution instance.

**class distutils.core.Command**

A `Command` class (or rather, an instance of one of its subclasses) implement a single distutils command.
10.2. distutils.ccompiler — CCompiler base class

This module provides the abstract base class for the `CCompiler` classes. A `CCompiler` instance can be used for all the compile and link steps needed to build a single project. Methods are provided to set options for the compiler — macro definitions, include directories, link path, libraries and the like.

This module provides the following functions.

**distutils.ccompiler.**

- **gen_lib_options**(compiler, library_dirs, runtime_library_dirs, libraries)
  Generate linker options for searching library directories and linking with specific libraries. `libraries` and `library_dirs` are, respectively, lists of library names (not filenames!) and search directories. Returns a list of command-line options suitable for use with some compiler (depending on the two format strings passed in).

- **gen_preprocess_options**(macros, include_dirs)
  Generate C pre-processor options (`-D`, `-U`, `-I`) as used by at least two types of compilers: the typical Unix compiler and Visual C++. `macros` is the usual thing, a list of 1- or 2-tuples, where `(name,)` means undefine (`-U`) macro `name`, and `(name, value)` means define (`-D`) macro `name` to `value`. `include_dirs` is just a list of directory names to be added to the header file search path (`-I`). Returns a list of command-line options suitable for either Unix compilers or Visual C++.

- **get_default_compiler**(osname, platform)
  Determine the default compiler to use for the given platform.
osname should be one of the standard Python OS names (i.e. the ones returned by os.name) and platform the common value returned by sys.platform for the platform in question.

The default values are os.name and sys.platform in case the parameters are not given.

distutils.ccompiler.new_compiler(plat=None, compiler=None, verbose=0, dry_run=0, force=0)

Factory function to generate an instance of some CCompiler subclass for the supplied platform/compiler combination. plat defaults to os.name (eg. 'posix', 'nt'), and compiler defaults to the default compiler for that platform. Currently only 'posix' and 'nt' are supported, and the default compilers are “traditional Unix interface” (UnixCCompiler class) and Visual C++ (MSVCCompiler class). Note that it’s perfectly possible to ask for a Unix compiler object under Windows, and a Microsoft compiler object under Unix—if you supply a value for compiler, plat is ignored.

distutils.ccompiler.show_compilers()

Print list of available compilers (used by the --help-compiler options to build, build_ext, build_clib).

class distutils.ccompiler.CCompiler([verbose=0, dry_run=0, force=0])

The abstract base class CCompiler defines the interface that must be implemented by real compiler classes. The class also has some utility methods used by several compiler classes.

The basic idea behind a compiler abstraction class is that each instance can be used for all the compile/link steps in building a single project. Thus, attributes common to all of those compile and link steps — include directories, macros to define, libraries to
link against, etc. — are attributes of the compiler instance. To allow for variability in how individual files are treated, most of those attributes may be varied on a per-compilation or per-link basis.

The constructor for each subclass creates an instance of the Compiler object. Flags are verbose (show verbose output), dry_run (don’t actually execute the steps) and force (rebuild everything, regardless of dependencies). All of these flags default to 0 (off). Note that you probably don’t want to instantiate CCompiler or one of its subclasses directly - use the distutils.CCompiler.new_compiler() factory function instead.

The following methods allow you to manually alter compiler options for the instance of the Compiler class.

add_include_dir(dir)
Add dir to the list of directories that will be searched for header files. The compiler is instructed to search directories in the order in which they are supplied by successive calls to add_include_dir().

set_include_dirs(dirs)
Set the list of directories that will be searched to dirs (a list of strings). Overrides any preceding calls to add_include_dir(); subsequent calls to add_include_dir() add to the list passed to set_include_dirs(). This does not affect any list of standard include directories that the compiler may search by default.

add_library(libname)
Add libname to the list of libraries that will be included in all links driven by this compiler object. Note that libname should *not* be the name of a file containing a library, but the name
of the library itself: the actual filename will be inferred by the linker, the compiler, or the compiler class (depending on the platform).

The linker will be instructed to link against libraries in the order they were supplied to `add_library()` and/or `set_libraries()`. It is perfectly valid to duplicate library names; the linker will be instructed to link against libraries as many times as they are mentioned.

**set_libraries(libnames)**
Set the list of libraries to be included in all links driven by this compiler object to `libnames` (a list of strings). This does not affect any standard system libraries that the linker may include by default.

**add_library_dir(dir)**
Add `dir` to the list of directories that will be searched for libraries specified to `add_library()` and `set_libraries()`. The linker will be instructed to search for libraries in the order they are supplied to `add_library_dir()` and/or `set_library_dirs()`.

**set_library_dirs(dirs)**
Set the list of library search directories to `dirs` (a list of strings). This does not affect any standard library search path that the linker may search by default.

**add_runtime_library_dir(dir)**
Add `dir` to the list of directories that will be searched for shared libraries at runtime.

**set_runtime_library_dirs(dirs)**
Set the list of directories to search for shared libraries at runtime to `dirs` (a list of strings). This does not affect any
standard search path that the runtime linker may search by default.

**define_macro**(name[, value=None])
Define a preprocessor macro for all compilations driven by this compiler object. The optional parameter *value* should be a string; if it is not supplied, then the macro will be defined without an explicit value and the exact outcome depends on the compiler used.

**undefine_macro**(name)
Undefine a preprocessor macro for all compilations driven by this compiler object. If the same macro is defined by **define_macro()** and undefined by **undefine_macro()** the last call takes precedence (including multiple redefinitions or undefinitions). If the macro is redefined/undefined on a per-compilation basis (ie. in the call to **compile()**), then that takes precedence.

**add_link_object**(object)
Add *object* to the list of object files (or analogues, such as explicitly named library files or the output of “resource compilers”) to be included in every link driven by this compiler object.

**set_link_objects**(objects)
Set the list of object files (or analogues) to be included in every link to *objects*. This does not affect any standard object files that the linker may include by default (such as system libraries).

The following methods implement methods for autodetection of compiler options, providing some functionality similar to GNU **autoconf**.
detect_language(sources)
Detect the language of a given file, or list of files. Uses the instance attributes language_map (a dictionary), and language_order (a list) to do the job.

find_library_file(dirs, lib[, debug=0])
Search the specified list of directories for a static or shared library file lib and return the full path to that file. If debug is true, look for a debugging version (if that makes sense on the current platform). Return None if lib wasn’t found in any of the specified directories.

has_function(funcname[, includes=None, include_dirs=None, libraries=None, library_dirs=None])
Return a boolean indicating whether funcname is supported on the current platform. The optional arguments can be used to augment the compilation environment by providing additional include files and paths and libraries and paths.

library_dir_option(dir)
Return the compiler option to add dir to the list of directories searched for libraries.

library_option(lib)
Return the compiler option to add lib to the list of libraries linked into the shared library or executable.

runtime_library_dir_option(dir)
Return the compiler option to add dir to the list of directories searched for runtime libraries.

set_executables(**args)
Define the executables (and options for them) that will be run to perform the various stages of compilation. The exact set of
executables that may be specified here depends on the compiler class (via the ‘executables’ class attribute), but most will have:

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compiler</td>
<td>the C/C++ compiler</td>
</tr>
<tr>
<td>linker_so</td>
<td>linker used to create shared objects and libraries</td>
</tr>
<tr>
<td>linker_exe</td>
<td>linker used to create binary executables</td>
</tr>
<tr>
<td>archiver</td>
<td>static library creator</td>
</tr>
</tbody>
</table>

On platforms with a command-line (Unix, DOS/Windows), each of these is a string that will be split into executable name and (optional) list of arguments. (Splitting the string is done similarly to how Unix shells operate: words are delimited by spaces, but quotes and backslashes can override this. See `distutils.util.split_quoted()`.)

The following methods invoke stages in the build process.

```python
compile(sources[, output_dir=None, macros=None, include_dirs=None, debug=0, extra_preargs=None, extra_postargs=None, depends=None])
```

Compile one or more source files. Generates object files (e.g. transforms a `.c` file to a `.o` file.)

`sources` must be a list of filenames, most likely C/C++ files, but in reality anything that can be handled by a particular compiler and compiler class (eg. `MSVCCompiler` can handle resource files in `sources`). Return a list of object filenames, one per source filename in `sources`. Depending on the implementation, not all source files will necessarily be compiled, but all corresponding object filenames will be returned.
If `output_dir` is given, object files will be put under it, while retaining their original path component. That is, `foo/bar.c` normally compiles to `foo/bar.o` (for a Unix implementation); if `output_dir` is `build`, then it would compile to `build/foo/bar.o`.

`macros`, if given, must be a list of macro definitions. A macro definition is either a `(name, value)` 2-tuple or a `(name,)` 1-tuple. The former defines a macro; if the value is `None`, the macro is defined without an explicit value. The 1-tuple case undefines a macro. Later definitions/redefinitions/undefinitions take precedence.

`include_dirs`, if given, must be a list of strings, the directories to add to the default include file search path for this compilation only.

`debug` is a boolean; if true, the compiler will be instructed to output debug symbols in (or alongside) the object file(s).

`extra_preargs` and `extra_postargs` are implementation-dependent. On platforms that have the notion of a command-line (e.g. Unix, DOS/Windows), they are most likely lists of strings: extra command-line arguments to prepend/append to the compiler command line. On other platforms, consult the implementation class documentation. In any event, they are intended as an escape hatch for those occasions when the abstract compiler framework doesn’t cut the mustard.

`depends`, if given, is a list of filenames that all targets depend on. If a source file is older than any file in depends, then the source file will be recompiled. This supports dependency tracking, but only at a coarse granularity.

Raises `CompileError` on failure.
create_static_lib(objects, output_libname[, output_dir=None, debug=0, target_lang=None])

Link a bunch of stuff together to create a static library file. The “bunch of stuff” consists of the list of object files supplied as objects, the extra object files supplied to add_link_object() and/or set_link_objects(), the libraries supplied to add_library() and/or set_libraries(), and the libraries supplied as libraries (if any).

output_libname should be a library name, not a filename; the filename will be inferred from the library name. output_dir is the directory where the library file will be put.

debug is a boolean; if true, debugging information will be included in the library (note that on most platforms, it is the compile step where this matters: the debug flag is included here just for consistency).

target_lang is the target language for which the given objects are being compiled. This allows specific linkage time treatment of certain languages.

 Raises LibError on failure.

link(target_desc, objects, output_filename[, output_dir=None, libraries=None, library_dirs=None, runtime_library_dirs=None, export_symbols=None, debug=0, extra_preargs=None, extra_postargs=None, build_temp=None, target_lang=None])

Link a bunch of stuff together to create an executable or shared library file.

The “bunch of stuff” consists of the list of object files supplied as objects. output_filename should be a filename. If output_dir is supplied, output_filename is relative to it (i.e.
**output_filename** can provide directory components if needed).

**libraries** is a list of libraries to link against. These are library names, not filenames, since they're translated into filenames in a platform-specific way (eg. *foo* becomes *libfoo.a* on Unix and *foo.lib* on DOS/Windows). However, they can include a directory component, which means the linker will look in that specific directory rather than searching all the normal locations.

**library_dirs**, if supplied, should be a list of directories to search for libraries that were specified as bare library names (ie. no directory component). These are on top of the system default and those supplied to *add_library_dir()* and/or *set_library_dirs()*.

**runtime_library_dirs** is a list of directories that will be embedded into the shared library and used to search for other shared libraries that *it* depends on at run-time. (This may only be relevant on Unix.)

**export_symbols** is a list of symbols that the shared library will export. (This appears to be relevant only on Windows.)

**debug** is as for *compile()* and *create_static_lib()*, with the slight distinction that it actually matters on most platforms (as opposed to *create_static_lib()* , which includes a **debug** flag mostly for form's sake).

**extra_preargs** and **extra_postargs** are as for *compile()* (except of course that they supply command-line arguments for the particular linker being used).

**target_lang** is the target language for which the given objects are being compiled. This allows specific linkage time treatment of certain languages.
Raises `LinkError` on failure.

```python
link_executable(objects, output_progname[, output_dir=None, libraries=None, library_dirs=None, runtime_library_dirs=None, debug=0, extra_preargs=None, extra_postargs=None, target_lang=None])
```

Link an executable. `output_progname` is the name of the file executable, while `objects` are a list of object filenames to link in. Other arguments are as for the `link()` method.

```python
link_shared_lib(objects, output_libname[, output_dir=None, libraries=None, library_dirs=None, runtime_library_dirs=None, export_symbols=None, debug=0, extra_preargs=None, extra_postargs=None, build_temp=None, target_lang=None])
```

Link a shared library. `output_libname` is the name of the output library, while `objects` is a list of object filenames to link in. Other arguments are as for the `link()` method.

```python
link_shared_object(objects, output_filename[, output_dir=None, libraries=None, library_dirs=None, runtime_library_dirs=None, export_symbols=None, debug=0, extra_preargs=None, extra_postargs=None, build_temp=None, target_lang=None])
```

Link a shared object. `output_filename` is the name of the shared object that will be created, while `objects` is a list of object filenames to link in. Other arguments are as for the `link()` method.

```python
preprocess(source[, output_file=None, macros=None, include_dirs=None, extra_preargs=None, extra_postargs=None])
```

Preprocess a single C/C++ source file, named in `source`. Output will be written to file named `output_file`, or `stdout` if
output_file not supplied. macros is a list of macro definitions as for compile(), which will augment the macros set with define_macro() and undefine_macro(). include_dirs is a list of directory names that will be added to the default list, in the same way as add_include_dir().

 Raises PreprocessError on failure.

The following utility methods are defined by the CCompiler class, for use by the various concrete subclasses.

 executable_filename(basename[, strip_dir=0, output_dir=''])
 Returns the filename of the executable for the given basename. Typically for non-Windows platforms this is the same as the basename, while Windows will get a .exe added.

 library_filename(libname[, lib_type='static', strip_dir=0, output_dir=''])
 Returns the filename for the given library name on the current platform. On Unix a library with lib_type of 'static' will typically be of the form liblibname.a, while a lib_type of 'dynamic' will be of the form liblibname.so.

 object_filenames(source_filenames[, strip_dir=0, output_dir=''])
 Returns the name of the object files for the given source files. source_filenames should be a list of filenames.

 shared_object_filename(basename[, strip_dir=0, output_dir=''])
 Returns the name of a shared object file for the given file name basename.
execute(func, args[, msg=None, level=1])
Invokes distutils.util.execute(). This method invokes a Python function func with the given arguments args, after logging and taking into account the dry_run flag.

spawn(cmd)
Invokes distutils.util.spawn(). This invokes an external process to run the given command.

mkpath(name[, mode=511])
Invokes distutils.dir_util.mkpath(). This creates a directory and any missing ancestor directories.

move_file(src, dst)
Invokes distutils.file_util.move_file(). Renames src to dst.

announce(msg[, level=1])
Write a message using distutils.log.debug().

warn(msg)
Write a warning message msg to standard error.

download_file(msg)
If the debug flag is set on this CCompiler instance, print msg to standard output, otherwise do nothing.
This module provides the `UnixCCompiler` class, a subclass of `CCompiler` that handles the typical Unix-style command-line C compiler:

- macros defined with `-Dname[=value]`
- macros undefined with `-Uname`
- include search directories specified with `-Idir`
- libraries specified with `-llib`
- library search directories specified with `-Ldir`
- compile handled by `cc` (or similar) executable with `-c` option: compiles `.c` to `.o`
- link static library handled by `ar` command (possibly with `ranlib`)
- link shared library handled by `cc -shared`
This module provides `MSVCCCompiler`, an implementation of the abstract `CCompiler` class for Microsoft Visual Studio. Typically, extension modules need to be compiled with the same compiler that was used to compile Python. For Python 2.3 and earlier, the compiler was Visual Studio 6. For Python 2.4 and 2.5, the compiler is Visual Studio .NET 2003. The AMD64 and Itanium binaries are created using the Platform SDK.

`MSVCCCompiler` will normally choose the right compiler, linker etc. on its own. To override this choice, the environment variables `DISTUTILS_USE_SDK` and `MSSdk` must be both set. `MSSdk` indicates that the current environment has been setup by the SDK’s `SetEnv.Cmd` script, or that the environment variables had been registered when the SDK was installed; `DISTUTILS_USE_SDK` indicates that the distutils user has made an explicit choice to override the compiler selection by `MSVCCCompiler`. 
This module provides `BorlandCCompiler`, a subclass of the abstract `CCompiler` class for the Borland C++ compiler.
This module provides the `CygwinCCompiler` class, a subclass of `UnixCCompiler` that handles the Cygwin port of the GNU C compiler to Windows. It also contains the `Mingw32CCompiler` class which handles the mingw32 port of GCC (same as cygwin in no-cygwin mode).
This module provides the EMXCCCompiler class, a subclass of UnixCCCompiler that handles the EMX port of the GNU C compiler to OS/2.
This module provides a few functions for creating archive files, such as tarballs or zipfiles.

```
from distutils.archive_util import make_archive, make_tarball, make_zipfile
```

### `make_archive(base_name, format[, root_dir=None, base_dir=None, verbose=0, dry_run=0])`

Create an archive file (eg. *zip* or *tar*). `base_name` is the name of the file to create, minus any format-specific extension; `format` is the archive format: one of *zip*, *tar*, *ztar*, or *gztar*. `root_dir` is a directory that will be the root directory of the archive; ie. we typically `chdir` into `root_dir` before creating the archive. `base_dir` is the directory where we start archiving from; ie. `base_dir` will be the common prefix of all files and directories in the archive. `root_dir` and `base_dir` both default to the current directory. Returns the name of the archive file.

### `make_tarball(base_name, base_dir[, compress='gzip', verbose=0, dry_run=0])`

Create an (optional compressed) archive as a tar file from all files in and under `base_dir`. `compress` must be *gzip* (the default), *compress*, *bzip2*, or None. Both *tar* and the compression utility named by `compress` must be on the default program search path, so this is probably Unix-specific. The output tar file will be named `base_dir.tar`, possibly plus the appropriate compression extension (*.gz*, *.bz2* or *.Z*). Return the output filename.

### `make_zipfile(base_name, base_dir[, compress=None, verbose=0, dry_run=0])`

Create a zip archive file from all files in and under `base_dir`. `compress` is the file compression utility to use (None or *gzip*, *compress*, *bzip2*, or None). Returns the filename of the archive file.
Create a zip file from all files in and under `base_dir`. The output zip file will be named `base_name + .zip`. Uses either the `zipfile` Python module (if available) or the InfoZIP `zip` utility (if installed and found on the default search path). If neither tool is available, raises `DistutilsExecError`. Returns the name of the output zip file.
10.9. distutils.dep_util — Dependency checking

This module provides functions for performing simple, timestamp-based dependency of files and groups of files; also, functions based entirely on such timestamp dependency analysis.

**distutils.dep_util.newer(source, target)**

Return true if *source* exists and is more recently modified than *target*, or if *source* exists and *target* doesn't. Return false if both exist and *target* is the same age or newer than *source*. Raise **DistutilsFileError** if *source* does not exist.

**distutils.dep_util.newer_pairwise(sources, targets)**

Walk two filename lists in parallel, testing if each source is newer than its corresponding target. Return a pair of lists (*sources*, *targets*) where source is newer than target, according to the semantics of **newer()**.

**distutils.dep_util.newer_group(sources, target[, missing='error'])**

Return true if *target* is out-of-date with respect to any file listed in *sources* In other words, if *target* exists and is newer than every file in *sources*, return false; otherwise return true. **missing** controls what we do when a source file is missing; the default ('error') is to blow up with an **OSError** from inside **os.stat()**; if it is 'ignore', we silently drop any missing source files; if it is 'newer', any missing source files make us assume that *target* is out-of-date (this is handy in “dry-run” mode: it'll make you pretend to carry out commands that wouldn't work because inputs are missing, but that doesn’t matter because you’re not actually going to run the commands).
This module provides functions for operating on directories and trees of directories.

```python
distutils.dir_util.mkpath(name[, mode=0777, verbose=0, dry_run=0])
```

Create a directory and any missing ancestor directories. If the directory already exists (or if `name` is the empty string, which means the current directory, which of course exists), then do nothing. Raise `DistutilsFileError` if unable to create some directory along the way (e.g. some sub-path exists, but is a file rather than a directory). If `verbose` is true, print a one-line summary of each `mkdir` to stdout. Return the list of directories actually created.

```python
distutils.dir_util.create_tree(base_dir, files[, mode=0777, verbose=0, dry_run=0])
```

Create all the empty directories under `base_dir` needed to put `files` there. `base_dir` is just the name of a directory which doesn’t necessarily exist yet; `files` is a list of filenames to be interpreted relative to `base_dir`. `base_dir` + the directory portion of every file in `files` will be created if it doesn’t already exist. `mode`, `verbose` and `dry_run` flags are as for `mkpath()`.

```python
distutils.dir_util.copy_tree(src, dst[, preserve_mode=1, preserve_times=1, preservesymlinks=0, update=0, verbose=0, dry_run=0])
```

Copy an entire directory tree `src` to a new location `dst`. Both `src` and `dst` must be directory names. If `src` is not a directory, raise
**DistutilsFileError.** If *dst* does not exist, it is created with *mkpath()*.

The end result of the copy is that every file in *src* is copied to *dst*, and directories under *src* are recursively copied to *dst*. Return the list of files that were copied or might have been copied, using their output name. The return value is unaffected by *update* or *dry_run*: it is simply the list of all files under *src*, with the names changed to be under *dst*.

*preserve_mode* and *preserve_times* are the same as for *distutils.file_util.copy_file()*; note that they only apply to regular files, not to directories. If *preserve_symlinks* is true, symlinks will be copied as symlinks (on platforms that support them!); otherwise (the default), the destination of the symlink will be copied. *update* and *verbose* are the same as for *copy_file()*.

Files in *src* that begin with `.nfs` are skipped (more information on these files is available in answer D2 of the [NFS FAQ page](https://example.com)).

*Changed in version 2.7.4:* NFS files are ignored.

```python
distutils.dir_util.remove_tree(directory[, verbose=0, dry_run=0])
```

Recursively remove *directory* and all files and directories underneath it. Any errors are ignored (apart from being reported to *sys.stdout* if *verbose* is true).
10.11. distutils.file_util — Single file operations

This module contains some utility functions for operating on individual files.

```
distutils.file_util.copy_file(src, dst[, preserve_mode=1, preserve_times=1, update=0, link=None, verbose=0, dry_run=0])
```

Copy file `src` to `dst`. If `dst` is a directory, then `src` is copied there with the same name; otherwise, it must be a filename. (If the file exists, it will be ruthlessly clobbered.) If `preserve_mode` is true (the default), the file's mode (type and permission bits, or whatever is analogous on the current platform) is copied. If `preserve_times` is true (the default), the last-modified and last-access times are copied as well. If `update` is true, `src` will only be copied if `dst` does not exist, or if `dst` does exist but is older than `src`.

`link` allows you to make hard links (using `os.link()`) or symbolic links (using `os.symlink()`) instead of copying: set it to 'hard' or 'sym'; if it is `None` (the default), files are copied. Don't set `link` on systems that don't support it: `copy_file()` doesn't check if hard or symbolic linking is available. It uses `_copy_file_contents()` to copy file contents.

Return a tuple `(dest_name, copied): dest_name` is the actual name of the output file, and `copied` is true if the file was copied (or would have been copied, if `dry_run` true).

```
distutils.file_util.move_file(src, dst[, verbose, dry_run])
```

Move file `src` to `dst`. If `dst` is a directory, the file will be moved into it with the same name; otherwise, `src` is just renamed to `dst`. 
Returns the new full name of the file.

**Warning:** Handles cross-device moves on Unix using `copy_file()`. What about other systems?

distutils.file_util.\texttt{write\_file}(filename, contents)

Create a file called \textit{filename} and write \textit{contents} (a sequence of strings without line terminators) to it.
This module contains other assorted bits and pieces that don't fit into any other utility module.

\texttt{distutils.util.get\_platform()}

Return a string that identifies the current platform. This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by \texttt{os.uname()})\texttt{), although the exact information included depends on the OS; eg. for IRIX the architecture isn’t particularly important (IRIX only runs on SGI hardware), but for Linux the kernel version isn’t particularly important.

Examples of returned values:

- \texttt{linux-i586}
- \texttt{linux-alpha}
- \texttt{solaris-2.6-sun4u}
- \texttt{irix-5.3}
- \texttt{irix64-6.2}

For non-POSIX platforms, currently just returns \texttt{sys.platform}.

For Mac OS X systems the OS version reflects the minimal version on which binaries will run (that is, the value of \texttt{MACOSX\_DEPLOYMENT\_TARGET} during the build of Python), not the OS version of the current system.

For universal binary builds on Mac OS X the architecture value reflects the universal binary status instead of the architecture of
the current processor. For 32-bit universal binaries the architecture is fat, for 64-bit universal binaries the architecture is fat64, and for 4-way universal binaries the architecture is universal. Starting from Python 2.7 and Python 3.2 the architecture fat3 is used for a 3-way universal build (ppc, i386, x86_64) and intel is used for a universal build with the i386 and x86_64 architectures.

Examples of returned values on Mac OS X:

- macosx-10.3-ppc
- macosx-10.3-fat
- macosx-10.5-universal
- macosx-10.6-intel

```
# convert_path

distutils.util.convert_path(pathname)
```

Return ‘pathname’ as a name that will work on the native filesystem, i.e. split it on ‘/’ and put it back together again using the current directory separator. Needed because filenames in the setup script are always supplied in Unix style, and have to be converted to the local convention before we can actually use them in the filesystem. Raises ValueError on non-Unix-ish systems if pathname either starts or ends with a slash.

```
# change_root

distutils.util.change_root(new_root, pathname)
```

Return pathname with new_root prepended. If pathname is relative, this is equivalent to os.path.join(new_root, pathname). Otherwise, it requires making pathname relative and then joining the two, which is tricky on DOS/Windows.

```
# check_environ

distutils.util.check_environ()
```

Ensure that ‘os.environ’ has all the environment variables we guarantee that users can use in config files, command-line options, etc. Currently this includes:
- **HOME** - user’s home directory (Unix only)
- **PLAT** - description of the current platform, including hardware and OS (see `get_platform()`)

```python
distutils.util.subst_vars(s, local_vars)
```
Perform shell/Perl-style variable substitution on s. Every occurrence of $ followed by a name is considered a variable, and variable is substituted by the value found in the `local_vars` dictionary, or in `os.environ` if it’s not in `local_vars`. `os.environ` is first checked/augmented to guarantee that it contains certain values: see `check_environ()`. Raise `ValueError` for any variables not found in either `local_vars` or `os.environ`.

Note that this is not a fully-fledged string interpolation function. A valid $variable can consist only of upper and lower case letters, numbers and an underscore. No { } or ( ) style quoting is available.

```python
distutils.util.split_quoted(s)
```
Split a string up according to Unix shell-like rules for quotes and backslashes. In short: words are delimited by spaces, as long as those spaces are not escaped by a backslash, or inside a quoted string. Single and double quotes are equivalent, and the quote characters can be backslash-escaped. The backslash is stripped from any two-character escape sequence, leaving only the escaped character. The quote characters are stripped from any quoted string. Returns a list of words.

```python
distutils.util.execute(func, args[, msg=None, verbose=0, dry_run=0])
```
Perform some action that affects the outside world (for instance, writing to the filesystem). Such actions are special because they are disabled by the `dry_run` flag. This method takes care of all
that bureaucracy for you; all you have to do is supply the function to call and an argument tuple for it (to embody the “external action” being performed), and an optional message to print.

**distutils.util.strtobool(val)**

Convert a string representation of truth to true (1) or false (0).

True values are `y`, `yes`, `t`, `true`, `on` and 1; false values are `n`, `no`, `f`, `false`, `off` and 0. Raises `ValueError` if `val` is anything else.

**distutils.util.byte_compile(py_files[, optimize=0, force=0, prefix=None, base_dir=None, verbose=1, dry_run=0, direct=None])**

Byte-compile a collection of Python source files to either `.pyc` or `.pyo` files in the same directory. `py_files` is a list of files to compile; any files that don’t end in `.py` are silently skipped. `optimize` must be one of the following:

- 0 - don’t optimize (generate `.pyc`)
- 1 - normal optimization (like `python -O`)
- 2 - extra optimization (like `python -OO`)

If `force` is true, all files are recompiled regardless of timestamps.

The source filename encoded in each bytecode file defaults to the filenames listed in `py_files`; you can modify these with `prefix` and `basedir`. `prefix` is a string that will be stripped off of each source filename, and `base_dir` is a directory name that will be prepended (after `prefix` is stripped). You can supply either or both (or neither) of `prefix` and `base_dir`, as you wish.

If `dry_run` is true, doesn’t actually do anything that would affect the filesystem.

Byte-compilation is either done directly in this interpreter process with the standard `py_compile` module, or indirectly by writing a
temporary script and executing it. Normally, you should let byte_compile() figure out to use direct compilation or not (see the source for details). The direct flag is used by the script generated in indirect mode; unless you know what you’re doing, leave it set to None.

distutils.util/rfc822_escape(header)
Return a version of header escaped for inclusion in an RFC 822 header, by ensuring there are 8 spaces space after each newline. Note that it does no other modification of the string.
This module provides the `Distribution` class, which represents the module distribution being built/installed/distributed.
This module provides the `Extension` class, used to describe C/C++ extension modules in setup scripts.
This module provides the DEBUG flag.
Provides exceptions used by the Distutils modules. Note that Distutils modules may raise standard exceptions; in particular, SystemExit is usually raised for errors that are obviously the end-user's fault (e.g. bad command-line arguments).

This module is safe to use in `from ... import *` mode; it only exports symbols whose names start with `Distutils` and end with `Error`. 
This module provides a wrapper around the standard `getopt` module that provides the following additional features:

- short and long options are tied together
- options have help strings, so `fancy_getopt()` could potentially create a complete usage summary
- options set attributes of a passed-in object
- boolean options can have “negative aliases” — eg. if `--quiet` is the “negative alias” of `--verbose`, then `--quiet` on the command line sets `verbose` to false.

**distutils.fancy_getopt**

`fancy_getopt(options, negative_opt, object, args)`

Wrapper function. `options` is a list of `(long_option, short_option, help_string)` 3-tuples as described in the constructor for `FancyGetopt`. `negative_opt` should be a dictionary mapping option names to option names, both the key and value should be in the `options` list. `object` is an object which will be used to store values (see the `getopt()` method of the `FancyGetopt` class). `args` is the argument list. Will use `sys.argv[1:]` if you pass `None` as `args`.

`wrap_text(text, width)`

Wraps `text` to less than `width` wide.

**class distutils.fancy_getopt.FancyGetopt([option_table=None])**

The `option_table` is a list of 3-tuples: `(long_option, short_option, help_string)`
If an option takes an argument, its *long_option* should have '=' appended; *short_option* should just be a single character, no ':' in any case. *short_option* should be *None* if a *long_option* doesn't have a corresponding *short_option*. All option tuples must have long options.

The **FancyGetopt** class provides the following methods:

**FancyGetopt**. **getopt**([*args=None, object=None]*)

Parse command-line options in *args*. Store as attributes on *object*.

If *args* is *None* or not supplied, uses `sys.argv[1:]`. If *object* is *None* or not supplied, creates a new *OptionDummy* instance, stores option values there, and returns a tuple (*args, object*). If *object* is supplied, it is modified in place and **getopt()** just returns *args*; in both cases, the returned *args* is a modified copy of the passed-in *args* list, which is left untouched.

**FancyGetopt**. **get_option_order**()

Returns the list of (*option, value*) tuples processed by the previous run of **getopt()**. Raises **RuntimeError** if **getopt()** hasn't been called yet.

**FancyGetopt**. **generate_help**([*header=None]*)

Generate help text (a list of strings, one per suggested line of output) from the option table for this **FancyGetopt** object.

If supplied, prints the supplied *header* at the top of the help.
This module provides the `FileList` class, used for poking about the filesystem and building lists of files.
10.19. distutils.log — Simple PEP 282-style logging
This module provides the `spawn()` function, a front-end to various platform-specific functions for launching another program in a subprocess. Also provides `find_executable()` to search the path for a given executable name.
10.21. distutils.sysconfig — System configuration information

The `distutils.sysconfig` module provides access to Python's low-level configuration information. The specific configuration variables available depend heavily on the platform and configuration. The specific variables depend on the build process for the specific version of Python being run; the variables are those found in the Makefile and configuration header that are installed with Python on Unix systems. The configuration header is called `pyconfig.h` for Python versions starting with 2.2, and `config.h` for earlier versions of Python.

Some additional functions are provided which perform some useful manipulations for other parts of the distutils package.

- **distutils.sysconfig.PREFIX**
  The result of `os.path.normpath(sys.prefix)`.

- **distutils.sysconfig.EXEC_PREFIX**
  The result of `os.path.normpath(sys.exec_prefix)`.

- **distutils.sysconfig.get_config_var(name)**
  Return the value of a single variable. This is equivalent to `get_config_vars().get(name)`.

- **distutils.sysconfig.get_config_vars(...)**
  Return a set of variable definitions. If there are no arguments, this returns a dictionary mapping names of configuration variables to values. If arguments are provided, they should be strings, and the return value will be a sequence giving the associated values. If a given name does not have a corresponding value, `None` will be included for that variable.
distutils.sysconfig.get_config_h_filename()
Return the full path name of the configuration header. For Unix, this will be the header generated by the configure script; for other platforms the header will have been supplied directly by the Python source distribution. The file is a platform-specific text file.

distutils.sysconfig.get_makefile_filename()
Return the full path name of the Makefile used to build Python. For Unix, this will be a file generated by the configure script; the meaning for other platforms will vary. The file is a platform-specific text file, if it exists. This function is only useful on POSIX platforms.

distutils.sysconfig.get_python_inc([plat_specific, prefix])
Return the directory for either the general or platform-dependent C include files. If plat_specific is true, the platform-dependent include directory is returned; if false or omitted, the platform-independent directory is returned. If prefix is given, it is used as either the prefix instead of PREFIX, or as the exec-prefix instead of EXEC_PREFIX if plat_specific is true.

distutils.sysconfig.get_python_lib([plat_specific, standard_lib, prefix])
Return the directory for either the general or platform-dependent library installation. If plat_specific is true, the platform-dependent include directory is returned; if false or omitted, the platform-independent directory is returned. If prefix is given, it is used as either the prefix instead of PREFIX, or as the exec-prefix instead of EXEC_PREFIX if plat_specific is true. If standard_lib is true, the directory for the standard library is returned rather than the directory for the installation of third-party extensions.
The following function is only intended for use within the `distutils` package.

`distutils.sysconfig.customize_compiler(compiler)`

Do any platform-specific customization of a `distutils.ccompiler.CCompiler` instance.

This function is only needed on Unix at this time, but should be called consistently to support forward-compatibility. It inserts the information that varies across Unix flavors and is stored in Python’s `Makefile`. This information includes the selected compiler, compiler and linker options, and the extension used by the linker for shared objects.

This function is even more special-purpose, and should only be used from Python’s own build procedures.

`distutils.sysconfig.set_python_build()`

Inform the `distutils.sysconfig` module that it is being used as part of the build process for Python. This changes a lot of relative locations for files, allowing them to be located in the build area rather than in an installed Python.
This module provides the `TextFile` class, which gives an interface to text files that (optionally) takes care of stripping comments, ignoring blank lines, and joining lines with backslashes.

```python
class distutils.text_file.TextFile([filename=None, file=None, **options])
```

This class provides a file-like object that takes care of all the things you commonly want to do when processing a text file that has some line-by-line syntax: strip comments (as long as `#` is your comment character), skip blank lines, join adjacent lines by escaping the newline (ie. backslash at end of line), strip leading and/or trailing whitespace. All of these are optional and independently controllable.

The class provides a `warn()` method so you can generate warning messages that report physical line number, even if the logical line in question spans multiple physical lines. Also provides `unreadline()` for implementing line-at-a-time lookahead.

`TextFile` instances are create with either `filename`, `file`, or both. `RuntimeError` is raised if both are `None`. `filename` should be a string, and `file` a file object (or something that provides `readline()` and `close()` methods). It is recommended that you supply at least `filename`, so that `TextFile` can include it in warning messages. If `file` is not supplied, `TextFile` creates its own using the `open()` built-in function.

The options are all boolean, and affect the values returned by
### readline()

<table>
<thead>
<tr>
<th>option name</th>
<th>description</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>strip_comments</td>
<td>strip from ' #' to end-of-line, as well as any whitespace leading up to the ' #':—unless it is escaped by a backslash</td>
<td>true</td>
</tr>
<tr>
<td>lstrip_ws</td>
<td>strip leading whitespace from each line before returning it</td>
<td>false</td>
</tr>
<tr>
<td>rstrip_ws</td>
<td>strip trailing whitespace (including line terminator!) from each line before returning it.</td>
<td>true</td>
</tr>
<tr>
<td>skip_blanks</td>
<td>skip lines that are empty <em>after</em> stripping comments and whitespace. (If both lstrip_ws and rstrip_ws are false, then some lines may consist of solely whitespace: these will <em>not</em> be skipped, even if skip_blanks is true.)</td>
<td>true</td>
</tr>
<tr>
<td>join_lines</td>
<td>if a backslash is the last non-newline character on a line after stripping comments and whitespace, join the following line to it to form one logical line; if N consecutive lines end with a backslash, then N+1 physical lines will be joined to form one logical line.</td>
<td>false</td>
</tr>
<tr>
<td>collapse_join</td>
<td>strip leading whitespace from lines that are joined to their predecessor; only matters if (join_lines and not lstrip_ws)</td>
<td>false</td>
</tr>
</tbody>
</table>

Note that since rstrip_ws can strip the trailing newline, the semantics of `readline()` must differ from those of the built-in file object's `readline()` method! In particular, `readline()` returns
None for end-of-file: an empty string might just be a blank line (or an all-whitespace line), if rstrip_ws is true but skip_blanks is not.

open(filename)
Open a new file filename. This overrides any file or filename constructor arguments.

close()
Close the current file and forget everything we know about it (including the filename and the current line number).

warn(msg[, line=None])
Print (to stderr) a warning message tied to the current logical line in the current file. If the current logical line in the file spans multiple physical lines, the warning refers to the whole range, such as "lines 3-5". If line is supplied, it overrides the current line number; it may be a list or tuple to indicate a range of physical lines, or an integer for a single physical line.

readline()
Read and return a single logical line from the current file (or from an internal buffer if lines have previously been “unread” with unreadline()). If the join_lines option is true, this may involve reading multiple physical lines concatenated into a single string. Updates the current line number, so calling warn() after readline() emits a warning about the physical line(s) just read. Returns None on end-of-file, since the empty string can occur if rstrip_ws is true but strip_blanks is not.

readlines()
Read and return the list of all logical lines remaining in the current file. This updates the current line number to the last line of the file.
unreadline(line)

Push line (a string) onto an internal buffer that will be checked by future readline() calls. Handy for implementing a parser with line-at-a-time lookahead. Note that lines that are “unread” with unreadline() are not subsequently re-cleansed (whitespace stripped, or whatever) when read with readline(). If multiple calls are made to unreadline() before a call to readline(), the lines will be returned most in most recent first order.
10.23. *distutils.version* — Version number classes
This module supplies the abstract base class `Command`.

```
class distutils.cmd.Command(dist)
```

Abstract base class for defining command classes, the “worker bees” of the Distutils. A useful analogy for command classes is to think of them as subroutines with local variables called `options`. The options are declared in `initialize_options()` and defined (given their final values) in `finalize_options()`, both of which must be defined by every command class. The distinction between the two is necessary because option values might come from the outside world (command line, config file, ...), and any options dependent on other options must be computed after these outside influences have been processed — hence `finalize_options()`. The body of the subroutine, where it does all its work based on the values of its options, is the `run()` method, which must also be implemented by every command class.

The class constructor takes a single argument `dist`, a `Distribution` instance.
10.25. Creating a new Distutils command

This section outlines the steps to create a new Distutils command.

A new command lives in a module in the `distutils.command` package. There is a sample template in that directory called `command_template`. Copy this file to a new module with the same name as the new command you're implementing. This module should implement a class with the same name as the module (and the command). So, for instance, to create the command `peel_banana` (so that users can run `setup.py peel_banana`), you'd copy `command_template` to `distutils/command/peel_banana.py`, then edit it so that it's implementing the class `peel_banana`, a subclass of `distutils.cmd.Command`.

Subclasses of `Command` must define the following methods.

`Command.initialize_options()`
Set default values for all the options that this command supports. Note that these defaults may be overridden by other commands, by the setup script, by config files, or by the command-line. Thus, this is not the place to code dependencies between options; generally, `initialize_options()` implementations are just a bunch of `self.foo = None` assignments.

`Command.finalize_options()`
Set final values for all the options that this command supports. This is always called as late as possible, i.e. after any option assignments from the command-line or from other commands have been done. Thus, this is the place to code option dependencies: if `foo` depends on `bar`, then it is safe to set `foo` from `bar` as long as `foo` still has the same value it was assigned
in `initialize_options()`.

**Command. run()**

A command's raison d'être: carry out the action it exists to perform, controlled by the options initialized in `initialize_options()`, customized by other commands, the setup script, the command-line, and config files, and finalized in `finalize_options()`. All terminal output and filesystem interaction should be done by `run()`.

**Command. sub_commands**

`sub_commands` formalizes the notion of a “family” of commands, e.g. install as the parent with sub-commands `install_lib`, `install_headers`, etc. The parent of a family of commands defines `sub_commands` as a class attribute; it's a list of 2-tuples `(command_name, predicate)`, with `command_name` a string and `predicate` a function, a string or `None`. `predicate` is a method of the parent command that determines whether the corresponding command is applicable in the current situation. (E.g. `install_headers` is only applicable if we have any C header files to install.) If `predicate` is `None`, that command is always applicable.

`sub_commands` is usually defined at the end of a class, because predicates can be methods of the class, so they must already have been defined. The canonical example is the `install` command.
10.26. distutils.command — Individual Distutils commands
10.27. distutils.command.bdist — Build a binary installer
10.28. distutils.command.bdist_packager —
Abstract base class for packagers
10.30. distutils.command.bdist_msi — Build a Microsoft Installer binary package

```python
class distutils.command.bdist_msi.bdist_msi

Builds a Windows Installer (.msi) binary package.

In most cases, the bdist_msi installer is a better choice than the bdist_wininst installer, because it provides better support for Win64 platforms, allows administrators to perform non-interactive installations, and allows installation through group policies.
```
10.31. distutils.command.bdist_rpm — Build a binary distribution as a Redhat RPM and SRPM
10.32. distutils.command.bdist_wininst — Build a Windows installer
10.33. distutils.command.sdist — Build a source distribution
distutils.command.build — Build all files of a package
10.35. distutils.command.build_clib — Build any C libraries in a package
distutils.command.build_ext — Build any extensions in a package
10.37. distutils.command.build_py — Build the .py/.pyc files of a package
10.38. distutils.command.build_scripts — Build the scripts of a package
This command removes the temporary files created by `build` and its subcommands, like intermediary compiled object files. With the `--all` option, the complete build directory will be removed.

Extension modules built in place will not be cleaned, as they are not in the build directory.
10.40. distutils.command.config — Perform package configuration
10.41. distutils.command.install — Install a package
10.42. distutils.command.install_data —
Install data files from a package
Install C/C++ header files from a package
10.44. distutils.command.install_lib — Install library files from a package
The `register` command registers the package with the Python Package Index. This is described in more detail in PEP 301.
The `check` command performs some tests on the meta-data of a package. For example, it verifies that all required meta-data are provided as the arguments passed to the `setup()` function.
Installing Python Modules (Legacy version)

Author: Greg Ward

See also:

Installing Python Modules
The up to date module installation documentations

This document describes the Python Distribution Utilities (“Distutils”) from the end-user’s point-of-view, describing how to extend the capabilities of a standard Python installation by building and installing third-party Python modules and extensions.

Note: This guide only covers the basic tools for building and distributing extensions that are provided as part of this version of Python. Third party tools offer easier to use and more secure alternatives. Refer to the quick recommendations section in the Python Packaging User Guide for more information.
Introduction

Although Python's extensive standard library covers many programming needs, there often comes a time when you need to add some new functionality to your Python installation in the form of third-party modules. This might be necessary to support your own programming, or to support an application that you want to use and that happens to be written in Python.

In the past, there has been little support for adding third-party modules to an existing Python installation. With the introduction of the Python Distribution Utilities (Distutils for short) in Python 2.0, this changed.

This document is aimed primarily at the people who need to install third-party Python modules: end-users and system administrators who just need to get some Python application running, and existing Python programmers who want to add some new goodies to their toolbox. You don't need to know Python to read this document; there will be some brief forays into using Python's interactive mode to explore your installation, but that's it. If you're looking for information on how to distribute your own Python modules so that others may use them, see the Distributing Python Modules (Legacy version) manual. Debugging the setup script may also be of interest.

Best case: trivial installation

In the best case, someone will have prepared a special version of the module distribution you want to install that is targeted specifically at your platform and is installed just like any other software on your platform. For example, the module developer might make an executable installer available for Windows users, an RPM package for users of RPM-based Linux systems (Red Hat, SuSE, Mandrake,
and many others), a Debian package for users of Debian-based Linux systems, and so forth.

In that case, you would download the installer appropriate to your platform and do the obvious thing with it: run it if it’s an executable installer, `rpm --install` it if it’s an RPM, etc. You don’t need to run Python or a setup script, you don’t need to compile anything—you might not even need to read any instructions (although it’s always a good idea to do so anyway).

Of course, things will not always be that easy. You might be interested in a module distribution that doesn’t have an easy-to-use installer for your platform. In that case, you’ll have to start with the source distribution released by the module’s author/maintainer. Installing from a source distribution is not too hard, as long as the modules are packaged in the standard way. The bulk of this document is about building and installing modules from standard source distributions.

### The new standard: Distutils

If you download a module source distribution, you can tell pretty quickly if it was packaged and distributed in the standard way, i.e. using the Distutils. First, the distribution’s name and version number will be featured prominently in the name of the downloaded archive, e.g. `foo-1.0.tar.gz` or `widget-0.9.7.zip`. Next, the archive will unpack into a similarly-named directory: `foo-1.0` or `widget-0.9.7`. Additionally, the distribution will contain a setup script `setup.py`, and a file named `README.txt` or possibly just `README`, which should explain that building and installing the module distribution is a simple matter of running one command from a terminal:

```
python setup.py install
```
For Windows, this command should be run from a command prompt window (Start ▶ Accessories):

```
setup.py install
```

If all these things are true, then you already know how to build and install the modules you’ve just downloaded: Run the command above. Unless you need to install things in a non-standard way or customize the build process, you don’t really need this manual. Or rather, the above command is everything you need to get out of this manual.
Standard Build and Install

As described in section The new standard: Distutils, building and installing a module distribution using the Distutils is usually one simple command to run from a terminal:

```
python setup.py install
```

Platform variations

You should always run the setup command from the distribution root directory, i.e. the top-level subdirectory that the module source distribution unpacks into. For example, if you've just downloaded a module source distribution `foo-1.0.tar.gz` onto a Unix system, the normal thing to do is:

```
gunzip -c foo-1.0.tar.gz | tar xf -  # unpacks into directory
cd foo-1.0
python setup.py install
```

On Windows, you’d probably download `foo-1.0.zip`. If you downloaded the archive file to `C:\Temp`, then it would unpack into `C:\Temp\foo-1.0`; you can use either an archive manipulator with a graphical user interface (such as WinZip) or a command-line tool (such as `unzip` or `pkunzip`) to unpack the archive. Then, open a command prompt window and run:

```
cd c:\Temp\foo-1.0
python setup.py install
```

Splitting the job up

Running `setup.py install` builds and installs all modules in one run.
If you prefer to work incrementally—especially useful if you want to customize the build process, or if things are going wrong—you can use the setup script to do one thing at a time. This is particularly helpful when the build and install will be done by different users—for example, you might want to build a module distribution and hand it off to a system administrator for installation (or do it yourself, with super-user privileges).

For example, you can build everything in one step, and then install everything in a second step, by invoking the setup script twice:

```bash
python setup.py build
python setup.py install
```

If you do this, you will notice that running the `install` command first runs the `build` command, which—in this case—quickly notices that it has nothing to do, since everything in the `build` directory is up-to-date.

You may not need this ability to break things down often if all you do is install modules downloaded off the ‘net, but it’s very handy for more advanced tasks. If you get into distributing your own Python modules and extensions, you’ll run lots of individual Distutils commands on their own.

---

**How building works**

As implied above, the `build` command is responsible for putting the files to install into a *build directory*. By default, this is `build` under the distribution root; if you’re excessively concerned with speed, or want to keep the source tree pristine, you can change the build directory with the `--build-base` option. For example:

```bash
python setup.py build --build-base=/path/to/pybuild/foo-1.0
```
(Or you could do this permanently with a directive in your system or personal Distutils configuration file; see section Distutils Configuration Files.) Normally, this isn't necessary.

The default layout for the build tree is as follows:

```plaintext
--- build/ --- lib/
or
--- build/ --- lib.<plat>/
    temp.<plat>/
```

where `<plat>` expands to a brief description of the current OS/hardware platform and Python version. The first form, with just a `lib` directory, is used for “pure module distributions”—that is, module distributions that include only pure Python modules. If a module distribution contains any extensions (modules written in C/C++), then the second form, with two `<plat>` directories, is used. In that case, the `temp.<plat>` directory holds temporary files generated by the compile/link process that don't actually get installed. In either case, the `lib` (or `lib.<plat>`) directory contains all Python modules (pure Python and extensions) that will be installed.

In the future, more directories will be added to handle Python scripts, documentation, binary executables, and whatever else is needed to handle the job of installing Python modules and applications.

## How installation works

After the `build` command runs (whether you run it explicitly, or the `install` command does it for you), the work of the `install` command is relatively simple: all it has to do is copy everything under `build/lib` (or `build/lib.<plat>`) to your chosen installation directory.

If you don’t choose an installation directory—i.e., if you just run `setup.py install`—then the `install` command installs to the standard
location for third-party Python modules. This location varies by
platform and by how you built/installed Python itself. On Unix (and
Mac OS X, which is also Unix-based), it also depends on whether
the module distribution being installed is pure Python or contains
extensions (“non-pure”):

<table>
<thead>
<tr>
<th>Platform</th>
<th>Standard installation location</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unix (pure)</td>
<td><code>prefix/lib/pythonX.Y/site-packages</code></td>
<td><code>/usr/local/lib/pythonX.Y/site-packages</code></td>
</tr>
<tr>
<td>Unix (non-pure)</td>
<td><code>exec-prefix/lib/pythonX.Y/site-packages</code></td>
<td><code>/usr/local/lib/pythonX.Y/site-packages</code></td>
</tr>
<tr>
<td>Windows</td>
<td><code>prefix\Lib\site-packages</code></td>
<td><code>C:\PythonXY\Lib\site-packages</code></td>
</tr>
</tbody>
</table>

Notes:

1. Most Linux distributions include Python as a standard part of the
system, so `prefix` and `exec-prefix` are usually both `/usr` on
Linux. If you build Python yourself on Linux (or any Unix-like
system), the default `prefix` and `exec-prefix` are `/usr/local`.
2. The default installation directory on Windows was `C:\Program
Files\Python` under Python 1.6a1, 1.5.2, and earlier.

`prefix` and `exec-prefix` stand for the directories that Python is
installed to, and where it finds its libraries at run-time. They are
always the same under Windows, and very often the same under
Unix and Mac OS X. You can find out what your Python installation
uses for `prefix` and `exec-prefix` by running Python in interactive
mode and typing a few simple commands. Under Unix, just type
`python` at the shell prompt. Under Windows, choose Start › Programs › Python X.Y › Python (command line). Once the
interpreter is started, you type Python code at the prompt. For
example, on my Linux system, I type the three Python statements
shown below, and get the output as shown, to find out my `prefix` and `exec-prefix`:

```
Python 2.4 (#26, Aug 7 2004, 17:19:02)
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.prefix
'/usr'
>>> sys.exec_prefix
'/usr'
```

A few other placeholders are used in this document: `x.y` stands for the version of Python, for example 2.7; `distname` will be replaced by the name of the module distribution being installed. Dots and capitalization are important in the paths; for example, a value that uses `python2.7` on UNIX will typically use `Python27` on Windows.

If you don’t want to install modules to the standard location, or if you don’t have permission to write there, then you need to read about alternate installations in section Alternate Installation. If you want to customize your installation directories more heavily, see section Custom Installation on custom installations.
Alternate Installation

Often, it is necessary or desirable to install modules to a location other than the standard location for third-party Python modules. For example, on a Unix system you might not have permission to write to the standard third-party module directory. Or you might wish to try out a module before making it a standard part of your local Python installation. This is especially true when upgrading a distribution already present: you want to make sure your existing base of scripts still works with the new version before actually upgrading.

The Distutils install command is designed to make installing module distributions to an alternate location simple and painless. The basic idea is that you supply a base directory for the installation, and the install command picks a set of directories (called an installation scheme) under this base directory in which to install files. The details differ across platforms, so read whichever of the following sections applies to you.

Note that the various alternate installation schemes are mutually exclusive: you can pass --user, or --home, or --prefix and --exec-prefix, or --install-base and --install-platbase, but you can’t mix from these groups.

Alternate installation: the user scheme

This scheme is designed to be the most convenient solution for users that don’t have write permission to the global site-packages directory or don’t want to install into it. It is enabled with a simple option:

```
python setup.py install --user
```
Files will be installed into subdirectories of `site.USER_BASE` (written as `userbase` hereafter). This scheme installs pure Python modules and extension modules in the same location (also known as `site.USER_SITE`). Here are the values for UNIX, including Mac OS X:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>modules</td>
<td><code>userbase/lib/pythonX.Y/site-packages</code></td>
</tr>
<tr>
<td>scripts</td>
<td><code>userbase/bin</code></td>
</tr>
<tr>
<td>data</td>
<td><code>userbase</code></td>
</tr>
<tr>
<td>C headers</td>
<td><code>userbase/include/pythonX.Y/distname</code></td>
</tr>
</tbody>
</table>

And here are the values used on Windows:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>modules</td>
<td><code>userbase\PythonXY\site-packages</code></td>
</tr>
<tr>
<td>scripts</td>
<td><code>userbase\Scripts</code></td>
</tr>
<tr>
<td>data</td>
<td><code>userbase</code></td>
</tr>
<tr>
<td>C headers</td>
<td><code>userbase\PythonXY\Include\distname</code></td>
</tr>
</tbody>
</table>

The advantage of using this scheme compared to the other ones described below is that the user site-packages directory is under normal conditions always included in `sys.path` (see `site` for more information), which means that there is no additional step to perform after running the `setup.py` script to finalize the installation.

The `build_ext` command also has a `--user` option to add `userbase/include` to the compiler search path for header files and `userbase/lib` to the compiler search path for libraries as well as to the runtime search path for shared C libraries (rpath).

### Alternate installation: the home scheme

The idea behind the “home scheme” is that you build and maintain a
personal stash of Python modules. This scheme’s name is derived from the idea of a “home” directory on Unix, since it’s not unusual for a Unix user to make their home directory have a layout similar to /usr/ or /usr/local/. This scheme can be used by anyone, regardless of the operating system they are installing for.

Installing a new module distribution is as simple as

```python
python setup.py install --home=<dir>
```

where you can supply any directory you like for the --home option. On Unix, lazy typists can just type a tilde (~); the install command will expand this to your home directory:

```python
python setup.py install --home=~
```

To make Python find the distributions installed with this scheme, you may have to modify Python’s search path or edit sitecustomize (see site) to call site.addsitedir() or edit sys.path.

The --home option defines the installation base directory. Files are installed to the following directories under the installation base as follows:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>modules</td>
<td>home/lib/python</td>
</tr>
<tr>
<td>scripts</td>
<td>home/bin</td>
</tr>
<tr>
<td>data</td>
<td>home</td>
</tr>
<tr>
<td>C headers</td>
<td>home/include/python/distname</td>
</tr>
</tbody>
</table>

(Mentally replace slashes with backslashes if you're on Windows.)

*Changed in version 2.4:* The --home option used to be supported only on Unix.
**Alternate installation: Unix (the prefix scheme)**

The “prefix scheme” is useful when you wish to use one Python installation to perform the build/install (i.e., to run the setup script), but install modules into the third-party module directory of a different Python installation (or something that looks like a different Python installation). If this sounds a trifle unusual, it is—that’s why the user and home schemes come before. However, there are at least two known cases where the prefix scheme will be useful.

First, consider that many Linux distributions put Python in `/usr`, rather than the more traditional `/usr/local`. This is entirely appropriate, since in those cases Python is part of “the system” rather than a local add-on. However, if you are installing Python modules from source, you probably want them to go in `/usr/local/lib/python2.X` rather than `/usr/lib/python2.X`. This can be done with

```
/usr/bin/python setup.py install --prefix=/usr/local
```

Another possibility is a network filesystem where the name used to write to a remote directory is different from the name used to read it: for example, the Python interpreter accessed as `/usr/local/bin/python` might search for modules in `/usr/local/lib/python2.X`, but those modules would have to be installed to, say, `/mnt/@server/export/lib/python2.X`. This could be done with

```
/usr/local/bin/python setup.py install --prefix=/mnt/@server/export
```

In either case, the `--prefix` option defines the installation base, and the `--exec-prefix` option defines the platform-specific installation base, which is used for platform-specific files. (Currently, this just
means non-pure module distributions, but could be expanded to C libraries, binary executables, etc.) If --exec-prefix is not supplied, it defaults to --prefix. Files are installed as follows:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python modules</td>
<td>prefix/lib/pythonX.Y/site-packages</td>
</tr>
<tr>
<td>extension modules</td>
<td>exec-prefix/lib/pythonX.Y/site-packages</td>
</tr>
<tr>
<td>scripts</td>
<td>prefix/bin</td>
</tr>
<tr>
<td>data</td>
<td>prefix</td>
</tr>
<tr>
<td>C headers</td>
<td>prefix/include/pythonX.Y/distname</td>
</tr>
</tbody>
</table>

There is no requirement that --prefix or --exec-prefix actually point to an alternate Python installation; if the directories listed above do not already exist, they are created at installation time.

Incidentally, the real reason the prefix scheme is important is simply that a standard Unix installation uses the prefix scheme, but with --prefix and --exec-prefix supplied by Python itself as sys.prefix and sys.exec_prefix. Thus, you might think you’ll never use the prefix scheme, but every time you run python setup.py install without any other options, you’re using it.

Note that installing extensions to an alternate Python installation has no effect on how those extensions are built: in particular, the Python header files (Python.h and friends) installed with the Python interpreter used to run the setup script will be used in compiling extensions. It is your responsibility to ensure that the interpreter used to run extensions installed in this way is compatible with the interpreter used to build them. The best way to do this is to ensure that the two interpreters are the same version of Python (possibly different builds, or possibly copies of the same build). (Of course, if your --prefix and --exec-prefix don’t even point to an alternate
Python installation, this is immaterial.)

**Alternate installation: Windows (the prefix scheme)**

Windows has no concept of a user's home directory, and since the standard Python installation under Windows is simpler than under Unix, the `--prefix` option has traditionally been used to install additional packages in separate locations on Windows.

```
python setup.py install --prefix="\Temp\Python"
```

to install modules to the `\Temp\Python` directory on the current drive.

The installation base is defined by the `--prefix` option; the `--exec-prefix` option is not supported under Windows, which means that pure Python modules and extension modules are installed into the same location. Files are installed as follows:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>modules</td>
<td><code>prefix\Lib\site-packages</code></td>
</tr>
<tr>
<td>scripts</td>
<td><code>prefix\Scripts</code></td>
</tr>
<tr>
<td>data</td>
<td><code>prefix</code></td>
</tr>
<tr>
<td>C headers</td>
<td><code>prefix\Include\distname</code></td>
</tr>
</tbody>
</table>
Custom Installation

Sometimes, the alternate installation schemes described in section Alternate Installation just don’t do what you want. You might want to tweak just one or two directories while keeping everything under the same base directory, or you might want to completely redefine the installation scheme. In either case, you’re creating a custom installation scheme.

To create a custom installation scheme, you start with one of the alternate schemes and override some of the installation directories used for the various types of files, using these options:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Override option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python modules</td>
<td>--install-purelib</td>
</tr>
<tr>
<td>extension modules</td>
<td>--install-platlib</td>
</tr>
<tr>
<td>all modules</td>
<td>--install-lib</td>
</tr>
<tr>
<td>scripts</td>
<td>--install-scripts</td>
</tr>
<tr>
<td>data</td>
<td>--install-data</td>
</tr>
<tr>
<td>C headers</td>
<td>--install-headers</td>
</tr>
</tbody>
</table>

These override options can be relative, absolute, or explicitly defined in terms of one of the installation base directories. (There are two installation base directories, and they are normally the same—they only differ when you use the Unix “prefix scheme” and supply different --prefix and --exec-prefix options; using --install-lib will override values computed or given for --install-purelib and --install-platlib, and is recommended for schemes that don’t make a difference between Python and extension modules.)

For example, say you’re installing a module distribution to your home directory under Unix—but you want scripts to go in ~/scripts rather than ~/bin. As you might expect, you can override this directory with
the --install-scripts option; in this case, it makes most sense to supply a relative path, which will be interpreted relative to the installation base directory (your home directory, in this case):

```bash
python setup.py install --home=~ --install-scripts=scripts
```

Another Unix example: suppose your Python installation was built and installed with a prefix of `/usr/local/python`, so under a standard installation scripts will wind up in `/usr/local/python/bin`. If you want them in `/usr/local/bin` instead, you would supply this absolute directory for the --install-scripts option:

```bash
python setup.py install --install-scripts=/usr/local/bin
```

(This performs an installation using the “prefix scheme,” where the prefix is whatever your Python interpreter was installed with—`/usr/local/python` in this case.)

If you maintain Python on Windows, you might want third-party modules to live in a subdirectory of `prefix`, rather than right in `prefix` itself. This is almost as easy as customizing the script installation directory —you just have to remember that there are two types of modules to worry about, Python and extension modules, which can conveniently be both controlled by one option:

```bash
python setup.py install --install-lib=Site
```

The specified installation directory is relative to `prefix`. Of course, you also have to ensure that this directory is in Python's module search path, such as by putting a `.pth` file in a site directory (see `site`). See section Modifying Python's Search Path to find out how to modify Python’s search path.

If you want to define an entire installation scheme, you just have to
supply all of the installation directory options. The recommended way to do this is to supply relative paths; for example, if you want to maintain all Python module-related files under `python` in your home directory, and you want a separate directory for each platform that you use your home directory from, you might define the following installation scheme:

```bash
python setup.py install --home=~/ 
  --install-purelib=python/lib 
  --install-platlib=python/lib.$PLAT 
  --install-scripts=python/scripts 
  --install-data=python/data
```

or, equivalently,

```bash
python setup.py install --home=~/python 
  --install-purelib=lib 
  --install-platlib='lib.$PLAT' 
  --install-scripts=scripts 
  --install-data=data
```

`$PLAT` is not (necessarily) an environment variable—it will be expanded by the Distutils as it parses your command line options, just as it does when parsing your configuration file(s).

Obviously, specifying the entire installation scheme every time you install a new module distribution would be very tedious. Thus, you can put these options into your Distutils config file (see section Distutils Configuration Files):

```
[install]
install-base=$HOME
install-purelib=python/lib
install-platlib=python/lib.$PLAT
install-scripts=python/scripts
install-data=python/data
```

or, equivalently,
Note that these two are not equivalent if you supply a different installation base directory when you run the setup script. For example,

```
python setup.py install --install-base=/tmp
```

would install pure modules to `/tmp/python/lib` in the first case, and to `/tmp/lib` in the second case. (For the second case, you probably want to supply an installation base of `/tmp/python`.)

You probably noticed the use of `$HOME` and `$PLAT` in the sample configuration file input. These are Distutils configuration variables, which bear a strong resemblance to environment variables. In fact, you can use environment variables in config files on platforms that have such a notion but the Distutils additionally define a few extra variables that may not be in your environment, such as `$PLAT`. (And of course, on systems that don't have environment variables, such as Mac OS 9, the configuration variables supplied by the Distutils are the only ones you can use.) See section Distutils Configuration Files for details.

## Modifying Python’s Search Path

When the Python interpreter executes an `import` statement, it searches for both Python code and extension modules along a search path. A default value for the path is configured into the Python binary when the interpreter is built. You can determine the path by importing the `sys` module and printing the value of `sys.path`.
The null string in `sys.path` represents the current working directory.

The expected convention for locally installed packages is to put them in the `.../site-packages/` directory, but you may want to install Python modules into some arbitrary directory. For example, your site may have a convention of keeping all software related to the web server under `/www`. Add-on Python modules might then belong in `/www/python`, and in order to import them, this directory must be added to `sys.path`. There are several different ways to add the directory.

The most convenient way is to add a path configuration file to a directory that’s already on Python’s path, usually to the `.../site-packages/` directory. Path configuration files have an extension of `.pth`, and each line must contain a single path that will be appended to `sys.path`. (Because the new paths are appended to `sys.path`, modules in the added directories will not override standard modules. This means you can’t use this mechanism for installing fixed versions of standard modules.)

Paths can be absolute or relative, in which case they’re relative to the directory containing the `.pth` file. See the documentation of the `site` module for more information.

A slightly less convenient way is to edit the `site.py` file in Python’s
standard library, and modify `sys.path`. `site.py` is automatically imported when the Python interpreter is executed, unless the `-s` switch is supplied to suppress this behaviour. So you could simply edit `site.py` and add two lines to it:

```python
import sys
sys.path.append('/www/python/')
```

However, if you reinstall the same major version of Python (perhaps when upgrading from 2.2 to 2.2.2, for example) `site.py` will be overwritten by the stock version. You’d have to remember that it was modified and save a copy before doing the installation.

There are two environment variables that can modify `sys.path`. `PYTHONHOME` sets an alternate value for the prefix of the Python installation. For example, if `PYTHONHOME` is set to `/www/python`, the search path will be set to `['', '/www/python/lib/pythonX.Y/', '/www/python/lib/pythonX.Y/plat-linux2', ...]`.

The `PYTHONPATH` variable can be set to a list of paths that will be added to the beginning of `sys.path`. For example, if `PYTHONPATH` is set to `/www/python:/opt/py`, the search path will begin with `['/www/python', '/opt/py']`. (Note that directories must exist in order to be added to `sys.path`; the `site` module removes paths that don’t exist.)

Finally, `sys.path` is just a regular Python list, so any Python application can modify it by adding or removing entries.
Distutils Configuration Files

As mentioned above, you can use Distutils configuration files to record personal or site preferences for any Distutils options. That is, any option to any command can be stored in one of two or three (depending on your platform) configuration files, which will be consulted before the command-line is parsed. This means that configuration files will override default values, and the command-line will in turn override configuration files. Furthermore, if multiple configuration files apply, values from “earlier” files are overridden by “later” files.

Location and names of config files

The names and locations of the configuration files vary slightly across platforms. On Unix and Mac OS X, the three configuration files (in the order they are processed) are:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Location and filename</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td><code>prefix/lib/pythonver/distutils/distutils.cfg</code></td>
<td>(1)</td>
</tr>
<tr>
<td>personal</td>
<td><code>~/.pydistutils.cfg</code></td>
<td>(2)</td>
</tr>
<tr>
<td>local</td>
<td><code>setup.cfg</code></td>
<td>(3)</td>
</tr>
</tbody>
</table>

And on Windows, the configuration files are:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Location and filename</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td><code>prefix\Lib\distutils\distutils.cfg</code></td>
<td>(4)</td>
</tr>
<tr>
<td>personal</td>
<td><code>%HOME%\pydistutils.cfg</code></td>
<td>(5)</td>
</tr>
<tr>
<td>local</td>
<td><code>setup.cfg</code></td>
<td>(3)</td>
</tr>
</tbody>
</table>

On all platforms, the “personal” file can be temporarily disabled by passing the `--no-user-cfg` option.
Notes:

1. Strictly speaking, the system-wide configuration file lives in the directory where the Distutils are installed; under Python 1.6 and later on Unix, this is as shown. For Python 1.5.2, the Distutils will normally be installed to \texttt{prefix/lib/python1.5/site-packages/distutils}, so the system configuration file should be put there under Python 1.5.2.

2. On Unix, if the \texttt{HOME} environment variable is not defined, the user’s home directory will be determined with the \texttt{getpwuid()} function from the standard \texttt{pwd} module. This is done by the \texttt{os.path.expanduser()} function used by Distutils.

3. I.e., in the current directory (usually the location of the setup script).

4. (See also note (1).) Under Python 1.6 and later, Python’s default “installation prefix” is \texttt{C:\Python}, so the system configuration file is normally \texttt{C:\Python\Lib\distutils\distutils.cfg}. Under Python 1.5.2, the default prefix was \texttt{C:\Program Files\Python}, and the Distutils were not part of the standard library—so the system configuration file would be \texttt{C:\Program Files\Python\distutils\distutils.cfg} in a standard Python 1.5.2 installation under Windows.

5. On Windows, if the \texttt{HOME} environment variable is not defined, \texttt{USERPROFILE} then \texttt{HOMEDRIVE} and \texttt{HOMEPATH} will be tried. This is done by the \texttt{os.path.expanduser()} function used by Distutils.

\section*{Syntax of config files}

The Distutils configuration files all have the same syntax. The config files are grouped into sections. There is one section for each Distutils command, plus a \texttt{global} section for global options that affect every command. Each section consists of one option per line, specified as
option=value.

For example, the following is a complete config file that just forces all commands to run quietly by default:

```
[global]
verbose=0
```

If this is installed as the system config file, it will affect all processing of any Python module distribution by any user on the current system. If it is installed as your personal config file (on systems that support them), it will affect only module distributions processed by you. And if it is used as the `setup.cfg` for a particular module distribution, it affects only that distribution.

You could override the default “build base” directory and make the `build*` commands always forcibly rebuild all files with the following:

```
[build]
build-base=blib
force=1
```

which corresponds to the command-line arguments

```
python setup.py build --build-base=blib --force
```

except that including the `build` command on the command-line means that command will be run. Including a particular command in config files has no such implication; it only means that if the command is run, the options in the config file will apply. (Or if other commands that derive values from it are run, they will use the values in the config file.)

You can find out the complete list of options for any command using the `--help` option, e.g.:
Python setup.py build --help

and you can find out the complete list of global options by using --help without a command:

Python setup.py --help

See also the “Reference” section of the “Distributing Python Modules” manual.
Building Extensions: Tips and Tricks

Whenever possible, the Distutils try to use the configuration information made available by the Python interpreter used to run the setup.py script. For example, the same compiler and linker flags used to compile Python will also be used for compiling extensions. Usually this will work well, but in complicated situations this might be inappropriate. This section discusses how to override the usual Distutils behaviour.

Tweaking compiler/linker flags

Compiling a Python extension written in C or C++ will sometimes require specifying custom flags for the compiler and linker in order to use a particular library or produce a special kind of object code. This is especially true if the extension hasn't been tested on your platform, or if you're trying to cross-compile Python.

In the most general case, the extension author might have foreseen that compiling the extensions would be complicated, and provided a Setup file for you to edit. This will likely only be done if the module distribution contains many separate extension modules, or if they often require elaborate sets of compiler flags in order to work.

A setup file, if present, is parsed in order to get a list of extensions to build. Each line in a Setup describes a single module. Lines have the following structure:

```plaintext
module ... [sourcefile ...] [cpparg ...] [library ...]
```

Let's examine each of the fields in turn.

- **module** is the name of the extension module to be built, and
should be a valid Python identifier. You can’t just change this in order to rename a module (edits to the source code would also be needed), so this should be left alone.

- **sourcefile** is anything that’s likely to be a source code file, at least judging by the filename. Filenames ending in `.c` are assumed to be written in C, filenames ending in `.c`, `.cc`, and `.cpp` are assumed to be C++, and filenames ending in `.m` or `.mm` are assumed to be in Objective C.
- **cpparg** is an argument for the C preprocessor, and is anything starting with `-I`, `-D`, `-U` or `-c`.
- **library** is anything ending in `.a` or beginning with `-l` or `-L`.

If a particular platform requires a special library on your platform, you can add it by editing the Setup file and running `python setup.py build`. For example, if the module defined by the line

```python
foo foomodule.c
```

must be linked with the math library `libm.a` on your platform, simply add `-lm` to the line:

```python
foo foomodule.c -lm
```

Arbitrary switches intended for the compiler or the linker can be supplied with the `-Xcompiler arg` and `-Xlinker arg` options:

```python
foo foomodule.c -Xcompiler -o32 -Xlinker -shared -lm
```

The next option after `-Xcompiler` and `-Xlinker` will be appended to the proper command line, so in the above example the compiler will be passed the `-o32` option, and the linker will be passed `-shared`. If a compiler option requires an argument, you’ll have to supply multiple `-Xcompiler` options; for example, to pass `-x c++` the Setup file would have to contain `-Xcompiler -x -Xcompiler c++`. 
Compiler flags can also be supplied through setting the `CFLAGS` environment variable. If set, the contents of `CFLAGS` will be added to the compiler flags specified in the `Setup` file.

Using non-Microsoft compilers on Windows

Borland/CodeGear C++

This subsection describes the necessary steps to use Distutils with the Borland C++ compiler version 5.5. First you have to know that Borland's object file format (OMF) is different from the format used by the Python version you can download from the Python or ActiveState Web site. (Python is built with Microsoft Visual C++, which uses COFF as the object file format.) For this reason you have to convert Python's library `python25.lib` into the Borland format. You can do this as follows:

```
coff2omf python25.lib python25_bcpp.lib
```

The `coff2omf` program comes with the Borland compiler. The file `python25.lib` is in the `Libs` directory of your Python installation. If your extension uses other libraries (`zlib, ...`) you have to convert them too.

The converted files have to reside in the same directories as the normal libraries.

How does Distutils manage to use these libraries with their changed names? If the extension needs a library (eg. `foo`) Distutils checks first if it finds a library with suffix `_bcpp` (eg. `foo_bcpp.lib`) and then uses this library. In the case it doesn't find such a special library it uses the default name (`foo.lib`).[1]

To let Distutils compile your extension with Borland C++ you now
have to type:

```plaintext
python setup.py build --compiler=bcpp
```

If you want to use the Borland C++ compiler as the default, you could specify this in your personal or system-wide configuration file for Distutils (see section Distutils Configuration Files.)

**See also:**

- **C++Builder Compiler**
  Information about the free C++ compiler from Borland, including links to the download pages.

- **Creating Python Extensions Using Borland's Free Compiler**
  Document describing how to use Borland's free command-line C++ compiler to build Python.

**GNU C / Cygwin / MinGW**

This section describes the necessary steps to use Distutils with the GNU C/C++ compilers in their Cygwin and MinGW distributions. [2]

For a Python interpreter that was built with Cygwin, everything should work without any of these following steps.

Not all extensions can be built with MinGW or Cygwin, but many can. Extensions most likely to not work are those that use C++ or depend on Microsoft Visual C extensions.

To let Distutils compile your extension with Cygwin you have to type:

```plaintext
python setup.py build --compiler=cygwin
```

and for Cygwin in no-cygwin mode [3] or for MinGW type:

```plaintext
python setup.py build --compiler=mingw32
```
If you want to use any of these options/compilers as default, you should consider writing it in your personal or system-wide configuration file for Distutils (see section Distutils Configuration Files.)

**Older Versions of Python and MinGW**

The following instructions only apply if you’re using a version of Python inferior to 2.4.1 with a MinGW inferior to 3.0.0 (with binutils-2.13.90-20030111-1).

These compilers require some special libraries. This task is more complex than for Borland’s C++, because there is no program to convert the library. First you have to create a list of symbols which the Python DLL exports. (You can find a good program for this task at https://sourceforge.net/projects/mingw/files/MinGW/Extension/pexports/)

```
pexports python25.dll >python25.def
```

The location of an installed `python25.dll` will depend on the installation options and the version and language of Windows. In a “just for me” installation, it will appear in the root of the installation directory. In a shared installation, it will be located in the system directory.

Then you can create from these information an import library for gcc.

```
/cygwin/bin/dlltool --dllname python25.dll --def python25.def -l
```

The resulting library has to be placed in the same directory as `python25.lib`. (Should be the `libs` directory under your Python installation directory.)
If your extension uses other libraries (zlib,...) you might have to convert them too. The converted files have to reside in the same directories as the normal libraries do.

See also:

Building Python modules on MS Windows platform with MinGW
   Information about building the required libraries for the MinGW environment.

Footnotes

[1] This also means you could replace all existing COFF-libraries with OMF-libraries of the same name.
[3] Then you have no POSIX emulation available, but you also don't need cygwin1.dll.
This document describes the Python Distribution Utilities (“Distutils”) from the module developer’s point of view, describing how to use the Distutils to make Python modules and extensions easily available to a wider audience with very little overhead for build/release/install mechanics.

**Note:** This guide only covers the basic tools for building and distributing extensions that are provided as part of this version of Python. Third party tools offer easier to use and more secure alternatives. Refer to the quick recommendations section in the Python Packaging User Guide for more information.

- **1. An Introduction to Distutils**
  - 1.1. Concepts & Terminology
  - 1.2. A Simple Example
  - 1.3. General Python terminology
  - 1.4. Distutils-specific terminology
- **2. Writing the Setup Script**
  - 2.1. Listing whole packages
  - 2.2. Listing individual modules
  - 2.3. Describing extension modules
  - 2.4. Relationships between Distributions and Packages
2.5. Installing Scripts
2.6. Installing Package Data
2.7. Installing Additional Files
2.8. Additional meta-data
2.9. Debugging the setup script

3. Writing the Setup Configuration File

4. Creating a Source Distribution
   4.1. Specifying the files to distribute
   4.2. Manifest-related options
   4.3. The MANIFEST.in template

5. Creating Built Distributions
   5.1. Creating dumb built distributions
   5.2. Creating RPM packages
   5.3. Creating Windows Installers
   5.4. Cross-compiling on Windows
   5.5. Vista User Access Control (UAC)

6. The Python Package Index (PyPI)
   6.1. PyPI overview
   6.2. Distutils commands
   6.3. PyPI package display

7. Examples
   7.1. Pure Python distribution (by module)
   7.2. Pure Python distribution (by package)
   7.3. Single extension module

8. Extending Distutils
   8.1. Integrating new commands
   8.2. Adding new distribution types

9. Command Reference
   9.1. Installing modules: the \texttt{install} command family

10. API Reference
    10.1. \texttt{distutils.core} — Core Distutils functionality
    10.2. \texttt{distutils.ccompiler} — CCompiler base class
    10.3. \texttt{distutils.unixccompiler} — Unix C Compiler
    10.4. \texttt{distutils.msvcompiler} — Microsoft Compiler
- 10.5. distutils.bcppcompiler — Borland Compiler
- 10.6. distutils.cygwincompiler — Cygwin Compiler
- 10.7. distutils.emxccompiler — OS/2 EMX Compiler
- 10.8. distutils.archive_util — Archiving utilities
- 10.9. distutils.dep_util — Dependency checking
- 10.10. distutils.dir_util — Directory tree operations
- 10.11. distutils.file_util — Single file operations
- 10.12. distutils.util — Miscellaneous other utility functions
- 10.13. distutils.dist — The Distribution class
- 10.14. distutils.extension — The Extension class
- 10.15. distutils.debug — Distutils debug mode
- 10.16. distutils.errors — Distutils exceptions
- 10.17. distutils.fancy_getopt — Wrapper around the standard getopt module
- 10.18. distutils.filelist — The FileList class
- 10.19. distutils.log — Simple PEP 282-style logging
- 10.20. distutils.spawn — Spawn a sub-process
- 10.21. distutils.sysconfig — System configuration information
- 10.22. distutils.text_file — The TextFile class
- 10.23. distutils.version — Version number classes
- 10.24. distutils.cmd — Abstract base class for Distutils commands
- 10.25. Creating a new Distutils command
- 10.26. distutils.command — Individual Distutils commands
- 10.27. distutils.command.bdist — Build a binary installer
- 10.28. distutils.command.bdist_packager — Abstract base class for packagers
- 10.29. distutils.command.bdist_dumb — Build a “dumb” installer
- 10.30. distutils.command.bdist_msi — Build a Microsoft Installer binary package
- 10.31. distutils.command.bdist_rpm — Build a binary distribution as a Redhat RPM and SRPM
- 10.32. distutils.command.bdist_wininst — Build a Windows installer
- 10.33. distutils.command.sdist — Build a source distribution
- 10.34. distutils.command.build — Build all files of a package
- 10.35. distutils.command.build_clib — Build any C libraries in a package
- 10.36. distutils.command.build_ext — Build any extensions in a package
- 10.37. distutils.command.build_py — Build the .py/.pyc files of a package
- 10.38. distutils.command.build_scripts — Build the scripts of a package
- 10.39. distutils.command.clean — Clean a package build area
- 10.40. distutils.command.config — Perform package configuration
- 10.41. distutils.command.install — Install a package
- 10.42. distutils.command.install_data — Install data files from a package
- 10.43. distutils.command.install_headers — Install C/C++ header files from a package
- 10.44. distutils.command.install_lib — Install library files from a package
- 10.45. distutils.command.install_scripts — Install script files from a package
- 10.46. distutils.command.register — Register a module
with the Python Package Index

- 10.47. distutils.command.check — Check the meta-data of a package
8. Extending Distutils

Distutils can be extended in various ways. Most extensions take the form of new commands or replacements for existing commands. New commands may be written to support new types of platform-specific packaging, for example, while replacements for existing commands may be made to modify details of how the command operates on a package.

Most extensions of the distutils are made within `setup.py` scripts that want to modify existing commands; many simply add a few file extensions that should be copied into packages in addition to `.py` files as a convenience.

Most distutils command implementations are subclasses of the `distutils.cmd.Command` class. New commands may directly inherit from `Command`, while replacements often derive from `Command` indirectly, directly subclassing the command they are replacing. Commands are required to derive from `Command`.
8.1. Integrating new commands

There are different ways to integrate new command implementations into distutils. The most difficult is to lobby for the inclusion of the new features in distutils itself, and wait for (and require) a version of Python that provides that support. This is really hard for many reasons.

The most common, and possibly the most reasonable for most needs, is to include the new implementations with your setup.py script, and cause the distutils.core.setup() function use them:

```python
from distutils.command.build_py import build_py as _build_py
from distutils.core import setup

class build_py(_build_py):
    #"""Specialized Python source builder."""

        # implement whatever needs to be different...

setup(cmdclass={'build_py': build_py},
     ...
)
```

This approach is most valuable if the new implementations must be used to use a particular package, as everyone interested in the package will need to have the new command implementation.

Beginning with Python 2.4, a third option is available, intended to allow new commands to be added which can support existing setup.py scripts without requiring modifications to the Python installation. This is expected to allow third-party extensions to provide support for additional packaging systems, but the commands can be used for anything distutils commands can be used for. A new configuration option, command_packages (command-line option --command-packages), can be used to specify additional packages to be
searched for modules implementing commands. Like all distutils options, this can be specified on the command line or in a configuration file. This option can only be set in the [global] section of a configuration file, or before any commands on the command line. If set in a configuration file, it can be overridden from the command line; setting it to an empty string on the command line causes the default to be used. This should never be set in a configuration file provided with a package.

This new option can be used to add any number of packages to the list of packages searched for command implementations; multiple package names should be separated by commas. When not specified, the search is only performed in the distutils.command package. When setup.py is run with the option --command-packages distcmds,buildcmds, however, the packages distutils.command, distcmds, and buildcmds will be searched in that order. New commands are expected to be implemented in modules of the same name as the command by classes sharing the same name. Given the example command line option above, the command bdist_openpkg could be implemented by the class distcmds.bdist_openpkg.bdist_openpkg or buildcmds.bdist_openpkg.bdist_openpkg.
8.2. Adding new distribution types

Commands that create distributions (files in the \texttt{dist/} directory) need to add \texttt{(command, filename)} pairs to \texttt{self.distribution.dist_files} so that \texttt{upload} can upload it to PyPI. The \texttt{filename} in the pair contains no path information, only the name of the file itself. In dry-run mode, pairs should still be added to represent what would have been created.
Index – Symbols

!=
  operator
%
  operator
% formatting
% interpolation
%=  
augmented assignment
%PATH%
&
  operator
&=
  augmented assignment
*
  in function calls
operator
statement, [1]
**=
  augmented assignment
*=  
augmented assignment
+
  operator
+=
  augmented assignment
-
  operator
--help
      command line option
-p, --pattern pattern
       unittest-discover command line option
-q
    compileall command line option
-Q <arg>
    command line option
-R
    command line option
-r N, --repeat=N
    timeit command line option
-R, --no-report
    trace command line option
-r, --report
    trace command line option
-S
    command line option
-s
    command line option
-s S, --setup=S
    timeit command line option
-s, --start-directory directory
    unittest-discover command line option
-s, --summary
    trace command line option
-t
    command line option
-t <zipfile>
    zipfile command line option
-t, --time
    timeit command line option
trace command line option
--ignore-dir=<dir>
trace command line option
--ignore-module=<mod>
trace command line option
--user-base
site command line option
--user-site
site command line option
--version
command line option
-3
command line option
-=
augmented assignment
-?
command line option
-B
command line option
-b, --buffer
unittest command line option
-c <command>
command line option
-c <zipfile> <source1> ... <sourceN>
zipfile command line option
-c, --catch
unittest command line option
-c, --clock
timeit command line option
-c, --count
trace command line option
-C, --coverdir=<dir>
trace command line option
-d
-t, --top-level-directory directory
unittest-discover command line option
-t, --trace
trace command line option
-T, --trackcalls
trace command line option
-U
command line option
-u
command line option
-V
command line option
-v
command line option
-v, --verbose
timeit command line option
unittest-discover command line option
-W arg
command line option
-X
command line option
-x
command line option
-x regex
compileall command line option
... .ini
file
.pdbrc
file
.pypirc file
.pythonrc.py
file
command line option
-d destdir
  compileall command line option
-E
  command line option
-e <zipfile> <output_dir>
  zipfile command line option
-f
  compileall command line option
-f, --failfast
  unittest command line option
-f, --file=<file>
  trace command line option
-g, --timing
  trace command line option
-h
  command line option
-h, --help
  timeit command line option
-i
  command line option
-i list
  compileall command line option
-J
  command line option
-l
  compileall command line option
-l <zipfile>
  zipfile command line option
-l, --listfuncs
  trace command line option
-m <module-name>
command line option augmented assignment
-m, --missing
trace command line option
-n N, --number=N
timeit command line option
-O
command line option
-OO
command line option
Index – _

__abs__() (in module operator)
    (object method)
__add__() (in module operator)
    (object method)
(rfc822.AddressList method)
__all__
    (optional module attribute)
    (package variable)
__and__() (in module operator)
    (object method)
__bases__ (class attribute), [1]
__builtin__
    module, [1], [2], [3], [4]
__builtin__ (module)
__builtins__
__call__() (object method), [1]
__class__ (instance attribute), [1]
__closure__ (function attribute)
__cmp__() (instance method)
    (object method)
__code__ (function attribute)
__coerce__() (object method)
__complex__() (object method)
__concat__() (in module operator)
__contains__()
__copy__() (copy protocol)
__debug__
__deepsCopy__() (copy protocol)
__defaults__ (function attribute)
__del__() (io.IOBase method)
__delattr__() (object method)
__delete__() (object method)
__delitem__() (email.message.Message method)
__dict__ (class attribute) (function attribute) (instance attribute), [1] (module attribute), [1] (object attribute)
__displayhook__ (in module sys)
__div__() (in module operator) (object method)
__divmod__() (object method)
_set__() (object method)
__setattr__() (object method), [1]
_setitem__() (email.message.Message method)
__setslice__() (in module operator) (object method)
__setstate__() (object method)
__slots__ (built-in variable)
__sub__() (in module operator) (object method) (rfc822.AddressList method)
__subclasscheck__() (class method)
__subclasses__() (class method)
__subclasshook__() (abc.ABCMeta method)
__truediv__() (in module operator) (object method) (rfc822.AddressList method)
__unicode__() (email.header.Header method)
<table>
<thead>
<tr>
<th>(module)</th>
<th>_name (ctypes.PyDLL attribute)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ge</strong>()</td>
<td>_objects (ctypes._CData attribute)</td>
</tr>
<tr>
<td>(in module operator)</td>
<td><em>pack</em> (ctypes.Structure attribute)</td>
</tr>
<tr>
<td>(instance method)</td>
<td>_parse() (gettext.NullTranslations method)</td>
</tr>
<tr>
<td>(object method)</td>
<td>_Pointer (class in ctypes)</td>
</tr>
<tr>
<td><strong>get</strong>() (object method)</td>
<td>_Py_c_diff (C function)</td>
</tr>
<tr>
<td><em>getattribute</em>_() (object method)</td>
<td>_Py_c_neg (C function)</td>
</tr>
<tr>
<td><strong>getattribute</strong>() (object method)</td>
<td>_Py_c_pow (C function)</td>
</tr>
<tr>
<td><strong>getinitargs</strong>() (object method)</td>
<td>_Py_c_prod (C function)</td>
</tr>
<tr>
<td><strong>getitem</strong>()</td>
<td>_Py_c_quot (C function)</td>
</tr>
<tr>
<td>(email.message.Message method)</td>
<td>_Py_c_sum (C function)</td>
</tr>
<tr>
<td>(in module operator)</td>
<td>_Py_NoneStruct (C variable)</td>
</tr>
<tr>
<td>(mailbox.Mailbox method)</td>
<td>_PyImport_FindExtension (C function)</td>
</tr>
<tr>
<td>(mapping object method)</td>
<td>_PyImport_Fini (C function)</td>
</tr>
<tr>
<td>(object method)</td>
<td>_PyImport_FixupExtension (C function)</td>
</tr>
<tr>
<td><strong>getnewargs</strong>() (object method)</td>
<td>_PyImport_Init (C function)</td>
</tr>
<tr>
<td><strong>getslice</strong>() (in module operator)</td>
<td>_PyObject_Del (C function)</td>
</tr>
<tr>
<td>(object method)</td>
<td>_PyObject_GC_TRACK (C function)</td>
</tr>
<tr>
<td><strong>getstate</strong>() (object method)</td>
<td>_PyObject_GC_UNTRACK (C function)</td>
</tr>
<tr>
<td><strong>globals</strong> (function attribute)</td>
<td>_PyObject_New (C function)</td>
</tr>
<tr>
<td><strong>gt</strong>() (in module operator)</td>
<td>_PyObject_NewVar (C function)</td>
</tr>
<tr>
<td>(instance method)</td>
<td>_PyString_Resize (C function)</td>
</tr>
<tr>
<td>(object method)</td>
<td>_PyTuple_Resize (C function)</td>
</tr>
<tr>
<td><strong>hash</strong>() (object method)</td>
<td>_quit() (FrameWork.Application method)</td>
</tr>
<tr>
<td><strong>hex</strong>() (object method)</td>
<td>_replace() (collections.somenamedtuple method)</td>
</tr>
<tr>
<td><strong>iadd</strong>() (in module operator)</td>
<td>_setroot()</td>
</tr>
<tr>
<td>(object method)</td>
<td>(xml.etree.ElementTree.ElementTree method)</td>
</tr>
<tr>
<td>(rfc822.AddressList method)</td>
<td>_SimpleCData (class in ctypes)</td>
</tr>
<tr>
<td></td>
<td>_start() (aetools.TalkTo method)</td>
</tr>
<tr>
<td></td>
<td>_structure() (in module email.iterators)</td>
</tr>
<tr>
<td></td>
<td><em>type</em> (ctypes._Pointer attribute)</td>
</tr>
<tr>
<td></td>
<td>(ctypes.Array attribute)</td>
</tr>
<tr>
<td></td>
<td>_urlopener (in module urllib)</td>
</tr>
<tr>
<td></td>
<td>_winreg (module)</td>
</tr>
<tr>
<td></td>
<td>_write() (wsgiref.handlers.BaseHandler method)</td>
</tr>
</tbody>
</table>
\_\_i\text{and}\_\_() \text{(in module operator)}
  \text{(object method)}
\_\_i\text{concat}\_\_() \text{(in module operator)}
\_\_i\text{div}\_\_() \text{(in module operator)}
  \text{(object method)}
\_\_i\text{floordiv}\_\_() \text{(in module operator)}
  \text{(object method)}
\_\_i\text{llshift}\_\_() \text{(in module operator)}
  \text{(object method)}
\_\_i\text{mod}\_\_() \text{(in module operator)}
  \text{(object method)}
\_\_i\text{import}\_\_\_\
  \text{built-in function}
\_\_i\text{import}\_\_() \text{(built-in function)}
\_\_i\text{mul}\_\_() \text{(in module operator)}
  \text{(object method)}
\_\_i\text{index}\_\_() \text{(in module operator)}
  \text{(object method)}
\_\_i\text{init}\_\_() \text{(logging.Handler method)}
  \text{(logging.logging.Formatter method)}
  \text{(object method), [1]}
\_\_i\text{instancecheck}\_\_() \text{(class method)}
\_\_i\text{int}\_\_() \text{(object method)}
\_\_i\text{inv}\_\_() \text{(in module operator)}
__invert__() (in module operator)
   (object method)
__ior__() (in module operator)
   (object method)
__ipow__() (in module operator)
   (object method)
__irepeat__() (in module operator)
__irshift__() (in module operator)
   (object method)
__isub__() (in module operator)
   (object method)
   (rfc822.AddressList method)
__iter__() (container method)
   (iterator method)
   (mailbox.Mailbox method)
   (object method)
   (unittest.TestSuite method)
__itruediv__() (in module operator)
   (object method)
__ixor__() (in module operator)
   (object method)
__le__() (in module operator)
   (instance method)
   (object method)
__len__() (email.message.Message method)
(mailbox.Mailbox method)
(mapping object method)
(object method)
(rfc822.AddressList method)
__loader__
__long__() (object method)
__lshift__() (in module operator)
   (object method)
__lt__() (in module operator)
   (instance method)
   (object method)
__main__
   module, [1], [2], [3], [4],
   [5], [6], [7]
__main__ (module)
__members__ (object attribute)
__metaclass__ (built-in variable)
__methods__ (object attribute)
__missing__()  
   (collections.defaultdict method)
   (object method)
__mod__() (in module operator)
   (object method)
__module__ (class attribute)
   (function attribute)
   (method attribute)
__mro__ (class attribute)
__mul__() (in module operator)
(object method)
__name__
(class attribute)
(definition attribute)
(function attribute)
(method attribute)
(module attribute), [1], [2]
__ne__() (email.charset.Charset method)
(email.header.Header method)
(in module operator)
(instance method)
(object method)
__neg__() (in module operator)
(object method)
__new__() (object method)
__nonzero__() (object method), [1]
__not__() (in module operator)
Index – A

A-LAW, [1]
a-LAW
a2b_base64() (in module binascii)
a2b_hex() (in module binascii)
a2b_hqx() (in module binascii)
a2b_qp() (in module binascii)
a2b_uu() (in module binascii)
abc (module)
ABCMeta (class in abc)
abort()
    (ftplib.FTP method)
    (in module os)
above() (curses.panel.Panel method)
abs
    built-in function, [1]
abs() (built-in function)
    (decimal.Context method)
    (in module operator)
abspath() (in module os.path)
abstract base class
AbstractBasicAuthHandler (class in urllib2)
AbstractDigestAuthHandler (class in urllib2)
AbstractFormatter (class in formatter)
abstractmethod() (in module abc)
abstractproperty() (in module abc)
AbstractWriter (class in formatter)
accept() (asyncore.dispatcher method)
    (multiprocessing.connection.Listener method)
    (socket.socket method)
accept2dyear (in module time)
access() (in module os)
acos() (in module cmath)
    (in module math)
acosh() (in module cmath)
    (in module math)
anchor_bgn() (htmlmili
anchor_end() (htmlmili
and
    bitwise
    operator, [1], [2]
and_() (in module op
annotate() (in module op
announce() (in module distutils
anonymous
    function
answer_challenge() (multiprocessing.connection)
any() (built-in function)
anydbm (module)
api_version (in module sys)
apop() (poplib.POP3
APPDATA
append() (array.array)
    (collections.deque)
    (email.header.Header)
    (imaplib.IMAP4)
    (list method)
    (msilib.CAB method)
    (pipes.Template)
    (xml.etree.Element)
appendChild() (xml.
appendleft() (collections
collection)
AppleEvents, [1]
aplesingle (module)
Application() (in module
application_uri() (in
apply
    built-in function, [
acquire() (logging.Handler method)
  (multiprocessing.Lock method)
  (multiprocessing.RLock method)
  (thread.lock method)
  (threading.Condition method)
  (threading.Lock method)
  (threading.RLock method)
  (threading.Semaphore method)
acquire_lock() (in module imp)
Action (class in argparse)
action (optparse.Option attribute)
ACTIONS (optparse.Option attribute)
activate_form() (fl.form method)
active_children() (in module multiprocessing)
active_count() (in module threading)
activeCount() (in module threading)
add() (decimal.Context method)
  (in module audioop)
  (in module operator)
  (mailbox.Mailbox method)
  (mailbox.Maildir method)
  (msilib.RadioButtonGroup method)
  (pstats.Stats method)
  (set method)
  (tarfile.TarFile method)
  (ttk.Notebook method)
add_alias() (in module email.charset)
add_argument() (argparse.ArgumentParser method)
add_argument_group()
  (argparse.ArgumentParser method)
add_box() (fl.form method)
add_browser() (fl.form method)
add_button() (fl.form method)
add_cgi_vars() (wsgiref.handlers.BaseHandler method)
add_charset() (in module email.charset)
apply (2to3 fixer)
apply() (built-in function)
  (multiprocessing.method)
apply_async() (multiprocessing.pool method)
architecture() (in module archive (zipimport.zip)
aRepr (in module repr)
argparse (module)
args (exceptions.Base)
  (functools.partial
targtypes (ctypes._Ft
argument
  call semantics
difference from p
  function
  function definition
ArgumentDefaultsHelpFormatter (class in
argument
  ArgumentDefault
ArgumentError
ArgumentParser (class in
argv (in module sys)
arithmetic
  conversion
  operation, binary
  operation, unary
ArithmeticError
array
  module
array (class in array)
Array (class in ctypes)
array (module)
Array() (in module m
  (in module multip
add_choice() (fl.form method)
add_clock() (fl.form method)
add_codec() (in module email.charset)
add_cookie_header() (cookielib.CookieJar method)
add_counter() (fl.form method)
add_data() (in module msilib)
   (urllib2.Request method)
add_dial() (fl.form method)
add_fallback() (gettext.NullTranslations method)
add_file() (msilib.Directory method)
add_flag() (mailbox.MaildirMessage method)
   (mailbox.MMDFMessage method)
   (mailbox.mboxMessage method)
add_flowing_data() (formatter.formatter method)
add_folder() (mailbox.Maildir method)
   (mailbox.MH method)
add_handler() (urllib2.OpenerDirector method)
add_header() (email.message.Message method)
   (urllib2.Request method)
   (wsgiref.headers.Headers method)
add_history() (in module readline)
add_hor_rule() (formatter.formatter method)
add_include_dir() (distutils.ccompiler.CCompiler method)
add_input() (fl.form method)
add_label() (mailbox.BabylMessage method)
add_label_data() (formatter.formatter method)
add_library() (distutils.ccompiler.CCompiler method)
add_library_dir() (distutils.ccompiler.CCompiler method)
add_lightbutton() (fl.form method)
add_line_break() (formatter.formatter method)
add_link_object() (distutils.ccompiler.CCompiler method)
add_literal_data() (formatter.formatter method)
add_menu() (fl.form method)

arrays
ArrayType (in module array)

article() (nntplib.NNTP method)
as
   import statement
   with statement
as_integer_ratio() (fl.form method)
AS_IS (in module formatter)
as_string() (email.message.Message method)
as_tuple() (decimal.Decimal method)
ascii() (in module curses.ascii)
   (in module future)
ASCII@ASCII, [1], [2]
ascii_letters (in module string)
ascii_lowercase (in module string)
ascii_uppercase (in module string)
asctime() (in module time)
asin() (in module cmath)
   (in module math)
asinh() (in module cmath)
AskFileForOpen() (in module EasyDialogs)
AskFileForSave() (in module EasyDialogs)
AskFolder() (in module EasyDialogs)
AskPassword() (in module EasyDialogs)
AskString() (in module EasyDialogs)
assert
   statement, [1]
assert_line_data() (formatter.formatter method)
assertAlmostEqual() (unittest.TestCase method)
assertDictContainsSubset() (unittest.TestCase method)
assertDictEqual() (unittest.TestCase method)
assertEquals() (unittest.TestCase method)
add_mutually_exclusive_group()
(argparse.ArgumentParser method)
add_option() (optparse.OptionParser method)
add_parent() (urllib2.BaseHandler method)
add_password() (urllib2.HTTPPasswordMgr method)
add_positioner() (fl.form method)
add_roundbutton() (fl.form method)
add_runtime_library_dir()
(distutils.ccompiler.CCompiler method)
add_section() (ConfigParser.RawConfigParser method)
add_sequence() (mailbox.MHMessage method)
add_slider() (fl.form method)
add_stream() (in module msilib)
add_subparsers() (argparse.ArgumentParser method)
add_suffix() (imputil.ImportManager method)
add_tables() (in module msilib)
add_text() (fl.form method)
add_timer() (fl.form method)
add_type() (in module mimetypes)
add_unredirected_header() (urllib2.Request method)
add_valslider() (fl.form method)
addch() (curses.window method)
addCleanup() (unittest.TestCase method)
addcomponent() (turtle.Shape method)
addError() (unittest.TestCase method)
addExpectedFailure() (unittest.TestCase method)
addFailure() (unittest.TestCase method)
addfile() (tarfile.TarFile method)
addFilter() (logging.Handler method)
   (logging.Logger method)
addHandler() (logging.Logger method)
addheader() (MimeWriter.MimeWriter method)
addinfo() (hotshot.Profile method)
addition
assertFalse() (unittest.TestCase method)
assertGreater() (unittest.TestCase method)
assertGreaterEqual() (unittest.TestCase method)
assertIn() (unittest.TestCase method)
AssertionError exception
assertions
debugging
assertIs() (unittest.TestCase method)
assertIsInstance() (unittest.TestCase method)
assertIsNone() (unittest.TestCase method)
assertIsNot() (unittest.TestCase method)
assertIsNotNone() (unittest.TestCase method)
assertItemsEqual() (unittest.TestCase method)
assertLess() (unittest.TestCase method)
assertLessEqual() (unittest.TestCase method)
assertListEqual() (unittest.TestCase method)
assertMultiLineEqual() (unittest.TestCase method)
assertNotAlmostEqual() (unittest.TestCase method)
assertNotEqual() (unittest.TestCase method)
assertNotIn() (unittest.TestCase method)
assertNotIsInstance() (unittest.TestCase method)
assertNotRegexpMatches() (unittest.TestCase method)
assertRaises() (unittest.TestCase method)
assertRaisesRegexp() (unittest.TestCase method)
assertRegexpMatches() (unittest.TestCase method)
asserts (2to3 fixer)
assertSequenceEqual() (unittest.TestCase method)
assertSetEqual() (unittest.TestCase method)
addLevelName() (in module logging)
addnstr() (curses.window method)
AddPackagePath() (in module modulefinder)
address (multiprocessing.connection.Listener attribute)
                        (multiprocessing.managers.BaseManager attribute)
address_family (SocketServer.BaseServer attribute)
address_string()
  (BaseHTTPServer.BaseHTTPRequestHandler method)
AddressList (class in rfc822)
addresslist (rfc822.AddressList attribute)
addressof() (in module ctypes)
addshape() (in module turtle)
addsitedir() (in module site)
addSkip() (unittest.TestResult method)
addstr() (curses.window method)
addSuccess() (unittest.TestResult method)
addTest() (unittest.TestSuite method)
addTests() (unittest.TestSuite method)
addTypeEqualityFunc() (unittest.TestCase method)
addUnexpectedSuccess() (unittest.TestResult method)
adjusted() (decimal.Decimal method)
adler32() (in module zlib)
ADPCM, Intel/DVI
adpcm2lin() (in module audioop)
aepack (module)
AEServer (class in MiniAEFrame)
AEText (class in aetypes)
aetools (module)
aetypes (module)
AF_INET (in module socket)
AF_INET6 (in module socket)
AF_UNIX (in module socket)
assertTrue() (unittest method)
assertTupleEqual() (unittest method)
assignment
                        attribute, [1]
augmented
class attribute
class instance att
extended slice
slice
slicing
statement, [1]
subscript
subscription
target list
AST (class in ast)
ast (module)
astimezone() (datetime method)
ASTVisitor (class in compiler.visitor)
async_chat (class in asynchat)
async_chat.ac_in_buffer_size (in module asynchat)
async_chat.ac_out_buffer_size (in module asynchat)
asynccevents() (Framework.Application method)
asyncchat (module)
asynccore (module)
AsyncResult (class in multiprocessing.pool)
AT (in module token)
atan() (in module cmath)
atan2() (in module math)
atanh() (in module math)
atexit (module)
atime (in module cd)
aifc (module)
aifc() (aifc.aifc method)
AIFF, [1]
aiff() (aifc.aifc method)
AIFF-C, [1]
AL
   module
AL (module)
al (module)
alarm() (in module signal)
alaw2lin() (in module audioop)
ALERT_DESCRIPTION_HANDSHAKE_FAILURE
   (in module ssl)
ALERT_DESCRIPTION_INTERNAL_ERROR
   (in module ssl)
algorithms_available (in module hashlib)
algorithms_guaranteed (in module hashlib)
alignment() (in module ctypes)
all() (built-in function)
al_errors (in module ftplib)
al_features (in module xml.sax.handler)
al_properties (in module xml.sax.handler)
allocate_lock() (in module thread)
allow_reuse_address (SocketServer.BaseServer
   attribute)
allowed_domains()
   (cookielib.DefaultCookiePolicy method)
al() (in module curses.ascii)
ALT_DIGITS (in module locale)
altsep (in module os)
altzone (in module time)
ALWAYS_TYPED_ACTIONS (optparse.Option
   attribute)
AMPER (in module token)
AMPEREQUAL (in module token)
atof() (in module locale)
atoi() (in module locale)
atol() (in module string)
attach() (email.message method)
AttlistDeclHandler()
   (xml.parsers.expat.xmlparser method)
attrgetter() (in module
   attrib (xml.etree.Element
   attribute), [1])
assignment, [1]
assignment, class
assignment, class
class
class instance
deletion
generic special
reference
special
AttributeError
   exception
attributes (xml.dom
   AttributesImpl (class
   AttributesNSImpl (cl
   attroff() (curses.wind
   attron() (curses.wind
   attrset() (curses.wind
   audio (in module cd)
Audio Interchange File
AUDIO_FILE_ENCODING_ADPCM_G721
   (in module sunau)
AUDIO_FILE_ENCODING_ADPCM_G722
   (in module sunau)
AUDIO_FILE_ENCODING_ADPCM_G723_3
(in module sunau)
AUDIO_FILE_ENCC
        (in module sunau)
AUDIO_FILE_ENCC
                module sunau)
AUDIO_FILE_ENCC
                    module sunau)
AUDIO_FILE_ENCC
                        sunau)
AUDIO_FILE_ENCC
                            module sunau)
AUDIO_FILE_ENCC
                                module sunau)
AUDIO_FILE_ENCC
                                    module sunau)
AUDIO_FILE_ENCC
                                        module sunau)
AUDIO_FILE_MAGIC
            (in module sunau)
AUDIODEV
audioop (module)
augmented assignment
auth() (ftplib.FTP_TLS method)
authenticate() (imaplib.IMAP4 method)
AuthenticationError
authenticator() (netrc.netrc method)
authkey (multiprocessing.Process attribute)
autoGIL (module)
AutoGILLError
avg() (in module audioop)
avgpp() (in module audioop)
b16decode() (in module base64)
b16encode() (in module base64)
b2a_base64() (in module binascii)
b2a_hex() (in module binascii)
b2a_hqx() (in module binascii)
b2a_qp() (in module binascii)
b2a_uu() (in module binascii)
b32decode() (in module base64)
b32encode() (in module base64)
b64decode() (in module base64)
b64encode() (in module base64)
Babyl (class in mailbox)
BabylMailbox (class in mailbox)
BabylMessage (class in mailbox)
back() (in module turtle)
back-quotes, [1]
BACKQUOTE (in module token)
backslash character
backslashreplace_errors() (in module codecs)
backward
   quotes, [1]
bkgdset() (curses.window method)
blank line
block
   code
blocked_domains() (cookielib.DefaultCookiePolicy method)
BlockingIOError
BLOCKSIZE (in module cd)
blocksize (in module sha)
BNF, [1]
body() (ntplib.NNTP method)
body_encode() (email.charset.Charset method)
body_encoding (email.charset.Charset attribute)
body_line_iterator() (in module email.iterators)
BOM (in module codecs)
BOM_BE (in module codecs)
BOM_LE (in module codecs)
BOM_UTF16 (in module codecs)
BOM_UTF16_BE (in module codecs)
BOM_UTF16_LE (in module codecs)
BOM_UTF32 (in module codecs)
BOM_UTF32_BE (in module codecs)
BOM_UTF32_LE (in module codecs)
BOM_UTF8 (in module codecs)
bool (built-in class)
   object, [1]
   operation
   operations, [1]
   type
   values
Boolean (class in aetypes)
backward() (in module turtle)
backward_compatible (in module imageop)
BadStatusLine
BadZipfile
Balloon (class in Tix)
bare except
base64
    encoding
    module
base64 (module)
BaseCGIHandler (class in wsgiref.handlers)
BaseCookie (class in Cookie)
BaseException
BaseHandler (class in urllib2)
    (class wsgiref.handlers)
BaseHTTPRequestHandler (class in BaseHTTPServer)
BaseHTTPServer (module)
BaseManager (class in multiprocessing.managers)
basename() (in module os.path)
BaseProxy (class in multiprocessing.managers)
BaseRequestHandler (class in SocketServer)
BaseServer (class in SocketServer)
basestring (2to3 fixer)
basestring() (built-in function)
basicConfig() (in module logging)

boolean() (in module xmlrpcclib)
BooleanType (in module types)
bootstrap() (in module ensurepip)
border() (curses.window method)
bottom() (curses.panel.Panel method)
bottom_panel() (in module curses.panel)
BoundaryError
BoundedSemaphore (class in multiprocessing)
BoundedSemaphore() (in module threading)
    (multiprocessing.managers.SyncManager method)
box() (curses.window method)
break
    statement, [1], [2], [3], [4]
break_anywhere() (bdb.Bdb method)
break_here() (bdb.Bdb method)
break_long_words (textwrap.TextWrapper attribute)
BREAK_LOOP (opcode)
break_on_hyphens (textwrap.TextWrapper attribute)
Breakpoint (class in bdb)
breakpoints
BROWSER, [1]
bsddb
    module, [1], [2], [3]
bsddb (module)
BsdDbShelf (class in shelve)
btopen() (in module bsddb)
buffer
    built-in function
    object, [1]
buffer (2to3 fixer)
    (io.TextIOBase attribute)
    (unittest.TestResult attribute)
buffer interface
buffer size, I/O
BasicContext (class in decimal)
Bastion (module)
Bastion() (in module Bastion)
BastionClass (class in Bastion)
baudrate() (in module curses)
bbox() (ttk.Treeview method)
bdb
   module
Bdb (class in bdb)
bdb (module)
BdbQuit
BDFL
bdist_msi (class in distutils.command.bdist_msi)
beep() (in module curses)
Beep() (in module winsound)
begin_fill() (in module turtle)
begin_poly() (in module turtle)
below() (curses.panel.Panel method)
Benchmarking
benchmarking
betavariate() (in module random)
bgcolor() (in module turtle)
bgn_group() (fl.form method)
bgpic() (in module turtle)
bias() (in module audioop)
bidirectional() (in module unicodedata)
BigEndianStructure (class in ctypes)
bin() (built-in function)
binary
buffer() (built-in function)
buffer_info() (array.array method)
buffer_size (xml.parsers.expat.xmlparser attribute)
buffer_text (xml.parsers.expat.xmlparser attribute)
buffer_used (xml.parsers.expat.xmlparser attribute)
BufferedIOBase (class in io)
BufferedReader (class in io)
BufferedRandom (class in io)
BufferedRWPair (class in io)
BufferedWriter (class in io)
BufferError
BufferingHandler (class in logging.handlers)
BufferTooShort
BUFFER (in module types), [1]
BUFSIZE (in module macostools)
bufsize() (ossaudiodev.oss_audio_device method)
BUILD_CLASS (opcode)
BUILD_LIST (opcode)
BUILD_MAP (opcode)
build_opener() (in module urllib2)
BUILD_SET (opcode)
BUILD_SLICE (opcode)
BUILD_TUPLE (opcode)
bldtools (module)
built-in
   method
   types
   built-in function
   __import__
   abs, [1]
   apply, [1], [2]
   buffer
   bytes
   call
arithmetic operation
bitwise operation
data, packing
Binary (class in msilib)
(class in xmlrpclib)
binary literal
binary semaphores
BINARY_ADD (opcode)
BINARY_AND (opcode)
BINARY_DIVIDE (opcode)
BINARY_FLOOR_DIVIDE (opcode)
BINARY_LSHIFT (opcode)
BINARY_MODULO (opcode)
BINARY_MULTIPLY (opcode)
BINARY_OR (opcode)
BINARY_POWER (opcode)
BINARY_RSHIFT (opcode)
BINARY_SUBSCR (opcode)
BINARY_SUBTRACT (opcode)
BINARY_TRUE_DIVIDE (opcode)
BINARY_XOR (opcode)
binascii (module)
bind (widgets)
bind() (asyncore.dispatcher method)
(socket.socket method)
bind_textdomain_codeset() (in module gettext)
binding
  global name
  name, [1], [2], [3], [4], [5]
bindtextdomain() (in module gettext)

chr, [1]
classmethod
cmp, [1], [2], [3]
coerce
compile, [1], [2], [3], [4]
complex, [1]
divmod, [1], [2]
eval, [1], [2], [3], [4], [5], [6], [7]
execfile, [1]
file
float, [1], [2], [3]
globals
hash, [1], [2]
help
hex
id
input, [1]
int, [1], [2]
len, [1], [2], [3], [4], [5], [6], [7], [8], [10], [11]
locals
long, [1], [2], [3]
max
min
object, [1]
oct
open, [1]
ord, [1], [2]
pow, [1], [2], [3], [4], [5]
range
raw_input, [1]
reload, [1], [2], [3]
repr, [1], [2], [3], [4], [5]
slice, [1], [2]
binhex
  module
binhex (module)
binhex() (in module binhex)
bisect (module)
bisect() (in module bisect)
bisect_left() (in module bisect)
bisect_right() (in module bisect)
bit_length() (int method)
  (long method)
bitmap() (msilib.Dialog method)
bitwise
  and
  operation, binary
  operation, unary
  operations
  or
  xor
bk() (in module turtle)
bkgd() (curses.window method)

staticmethod
str, [1], [2], [3]
tuple, [1]
type, [1], [2], [3]
unichr
unicode, [1], [2], [3]
xrange
built-in method
call
  object, [1]
builtin_module_names (in module sys)
BuiltinFunctionType (in module types)
BuiltinImporter (class in imputil)
BuiltinMethodType (in module types)
builtins
  module
ButtonBox (class in Tix)
bye() (in module turtle)
byref() (in module ctypes)
byte
byte-code
  file, [1], [2]
byte_compile() (in module distutils.util)
bytearray
  object, [1]
bytearray (built-in class)
bytecode, [1]
byteorder (in module sys)
bytes
  built-in function
bytes (uuid.UUID attribute)
bytes-like object
bytes_le (uuid.UUID attribute)
BytesIO (class in io)
byteswap() (array.array method)
bz2 (module)
BZ2Compressor (class in bz2)
BZ2Decompressor (class in bz2)
BZ2File (class in bz2)
Index – C

C
  language, [1], [2], [3], [4]
  structures
  c_bool (class in ctypes)
  C_BUILTIN (in module imp)
  c_byte (class in ctypes)
  c_char (class in ctypes)
  c_char_p (class in ctypes)
  c_double (class in ctypes)
  C_EXTENSION (in module imp)
  c_float (class in ctypes)
  c_int (class in ctypes)
  c_int16 (class in ctypes)
  c_int32 (class in ctypes)
  c_int64 (class in ctypes)
  c_int8 (class in ctypes)
  c_long (class in ctypes)
  c_longdouble (class in ctypes)
  c_longlong (class in ctypes)
  c_short (class in ctypes)
  c_size_t (class in ctypes)
  c_ssize_t (class in ctypes)
  c_ubyte (class in ctypes)
  c_uint (class in ctypes)
  c_uint16 (class in ctypes)
  c_uint32 (class in ctypes)
  c_uint64 (class in ctypes)
  c_uint8 (class in ctypes)
  c_ulong (class in ctypes)
  c_ulonglong (class in ctypes)
  c_ushort (class in ctypes)
  c_void_p (class in ctypes)
  c_wchar (class in ctypes)
  c_wchar_p (class in ctypes)
  CAB (class in msilib)
  code object, [1], [2], [3]
  code() (in module new)
  Codecs
    decode
    encode
  codecs (module)
  coded_value (Cookie.Ms)
  codeop (module)
  codepoint2name (in mod
  CODESET (in module lo
  coding
    style
  coerce
    built-in function
  coerce() (built-in functi
  coercion
  col_offset (ast.AST attri
  collapse_rfc2231_value(email.utils)
  collect() (in module gc)
  collect_incoming_data() (asynchat.async_chat method)
  collections (module)
  COLON (in module token
  color() (in module fl)
  (in module turtle)
  color_content() (in modu
  color_pair() (in module c
  colormode() (in module t
  ColorPicker (module)
  colorsys (module)
  column() (ttk.Treeview m
  COLUMNS, [1]
  combinations() (in modu
CacheFTPHandler (class in urllib2)
calcsise() (in module struct)
Calendar (class in calendar)
calendar (module)
calendar() (in module calendar)
call
    built-in function
    built-in method
    class instance
    class object, [1], [2], [3]
    function, [1], [2]
    instance, [1]
    method
    procedure
    user-defined function
call() (dl.dl method)
    (in module subprocess)
    CALL_FUNCTION (opcode)
    CALL_FUNCTION_KW (opcode)
    CALL_FUNCTION_VAR (opcode)
    CALL_FUNCTION_VAR_KW (opcode)
call_tracing() (in module sys)
callable
    object, [1]
callable (2to3 fixer)
Callable (class in collections)
callable() (built-in function)
CallableProxyType (in module weakref)
callback (optparse.Option attribute)
callback() (MiniAEFrame.AEServer method)
callback_args (optparse.Option attribute)
callback_kwargs (optparse.Option attribute)
CalledProcessError
calloc()
can_change_color() (in module curses)
can_fetch() (robotparser.RobotFileParser method)
combinations_with_replacement() (in module itertools)
combine() (datetime.datetime class)
combinig() (in module unicodedata)
ComboBox (class in Tix)
Comboxbox (class in ttk)
comma
    trailing, [1]
COMMA (in module token)
command
    command line
    command line option
        --help
        --version
        -3
        -?
        -B
        -E
        -J
        -O
        -OO
        -Q <arg>
        -R
        -S
        -U
        -V
        -W arg
        -X
        -c <command>
        -d
        -h
        -i
cancel() (sched.scheduler method)
   (threading.Timer method)
cancel_join_thread() (multiprocessing.Queue method)
CannotSendHeader
CannotSendRequest
canonic() (bdb.Bdb method)
canonical() (decimal.Context method)
   (decimal.Decimal method)
capitalize() (in module string)
   (str method)
Capsule
   object
captured_stdout() (in module test.test_support)
captureWarnings() (in module logging)
capwords() (in module string)
Carbon.AE (module)
Carbon.AH (module)
Carbon.App (module)
Carbon.Appearance (module)
Carbon.CarbonEvents (module)
Carbon.CarbonEvt (module)
Carbon.CF (module)
Carbon.CG (module)
Carbon.Cm (module)
Carbon.Components (module)
Carbon.ControlAccessor (module)
Carbon.Controls (module)
Carbon.CoreFoundation (module)
Carbon.CoreGraphics (module)
Carbon.Ctl (module)
Carbon.Dialogs (module)
Carbon.Dlg (module)
Carbon.Drag (module)
Carbon.Dragconst (module)
Carbon.Events (module)
CarbonEvt (module)

-m <module-name>
-s
-t
-u
-v
-x
CommandCompiler (class)
commands (module)
comment
(cookielib.Cookie attribute)
COMMENT (in module tokenize)
comment_url (cookielib.Cookie attribute)
commenters (shlex.shlex attribute)
CommentHandler()
(xml.parsers.expat.xmlproc)
commit() (msilib.CAB method)
Commit() (msilib.Database method)
commit() (sqlite3.Connection method)
common (filecmp.dircmp attribute)
Common Gateway Interface
common_dirs (filecmp.dircmp attribute)
common_files (filecmp.dircmp attribute)
common_funny (filecmp.dircmp attribute)
common_types (in module mimetypes)
commonprefix() (in module os.path)
communicate() (subprocess.Popen method)
compare() (decimal.Decimal method)

(difflib.Differ method)
compare_digest() (in module hashlib)
COMPARE_OP (opcode)
compare_signal() (decimal.Decimal method)
Carbon.File (module)
Carbon.Files (module)
Carbon.Fm (module)
Carbon.Folder (module)
Carbon.Folders (module)
Carbon.Fonts (module)
Carbon.Help (module)
Carbon.IBCarbon (module)
Carbon.IBCarbonRuntime (module)
Carbon.Icns (module)
Carbon.Icons (module)
Carbon.Launch (module)
Carbon.LaunchServices (module)
Carbon.List (module)
Carbon.Lists (module)
Carbon.MacHelp (module)
Carbon.MediaDescr (module)
Carbon.Menu (module)
Carbon.Menus (module)
Carbon.Mlte (module)
Carbon.OSA (module)
Carbon.OSAconst (module)
Carbon.Qd (module)
Carbon.Qdoffs (module)
Carbon.QDOffscreen (module)
Carbon.Qt (module)
Carbon.QuickDraw (module)
Carbon.QuickTime (module)
Carbon.Res (module)
Carbon.Resources (module)
Carbon.Scrap (module)
Carbon.Snd (module)
Carbon.Sound (module)
Carbon.TE (module)
Carbon.TextEdit (module)
Carbon.Win (module)
Carbon.Windows (module)
cast() (in module ctypes)
cast() (in module nis)

compare_total() (decimal.Context method)
compare_total_mag() (decimal.Decimal method)
comparing objects
comparison operator
string
Comparison (class in ae
COMPARISON_FLAGS
comparisons, [1]
chaining, [1]
compile built-in function, [1], [ Compile (class in codeop)
compile() (built-in function)
(distutils.ccompiler.C
(in module compiler)
(in module py_compi
(in module re)
(parser.ST method)
compile_command() (in
(in module codeop)
compile_dir() (in module
compile_file() (in module
compile_path() (in modu
compileall module
calculateall (module)
compileall command line
-d destdir
-f
-i list
-l
catalog (in module cd)
catch_warnings (class in warnings)
category() (in module unicodedata)
cbreak() (in module curses)
CC
CCompiler (class in distutils.ccompiler)
cc (module)
CDLL (class in ctypes)
CDROM (in module cd)
ceil() (in module math), [1]
center() (in module string)
    (str method)
CERT_NONE (in module ssl)
CERT_OPTIONAL (in module ssl)
CERT_REQUIRED (in module ssl)
cert_store_stats() (ssl.SSLContext method)
cert_time_to_seconds() (in module ssl)
CertificateError
certificates
CFLAGS, [1], [2]
cfmfile (module)
CFUNCTYPE() (in module ctypes)

CGI
    debugging
    exceptions
    protocol
    security
    tracebacks
cgi (module)
cgi_directories
    (CGIHTTPRequestHandler.
        attribute)
CGIHandler (class in wsgiref.handlers)
CGIHTTPRequestHandler (class in
    CGIHTTPServer)
CGIHTTPServer
    module
CGIHTTPServer (module)

-q
-x regex
directory ...
file ...
compileFile() (in module
compiler (module)
compiler.ast (module)
compiler.visitor (module)
compilest() (in module p
    complete() (rlcompleter.C
    complete_statement() (in
completedefault() (cmd.C

complex
    built-in function, [1]
literal
    number
    object
complex (built-in class)
Complex (class in numb
complex number
    literals
    object, [1]
ComplexType (in module
ComponentItem (class in

compound
    statement

comprehensions
    list, [1]
compress() (bz2.BZ2Compr
    (in module bz2)
    (in module itertools)
    (in module jpeg)
    (in module zlib)
    (zlib.Compress meth
compress_size (zipfile.Z
compress_type (zipfile.Z
cgitb (module)
CGIXMLRPCRequestHandler (class in SimpleXMLRPCServer)
chain() (in module itertools)
chaining
    comparisons, [1]
change_root() (in module distutils.util)
CHANNEL_BINDING_TYPES (in module ssl)
channels() (ossaudiodev.oss_audio_device method)
CHAR_MAX (in module locale)
character, [1], [2], [3]
character set
CharacterDataHandler()
(xml.parsers.expat.xmlparser method)
characters() (xml.sax.handler.ContentHandler method)
characters_written (io.BlockingIOError attribute)
charbufferproc (C type)
Charset (class in email.charset)
CHARSET (in module mimify)
charset() (gettext.NullTranslations method)
chdir() (in module os)
check() (imaplib.IMAP4 method)
   (in module tabnanny)
check_call() (in module subprocess)
check_environ() (in module distutils.util)
check_forms() (in module fl)
check_hostname (ssl.SSLContext attribute)
check_output() (doctest.OutputChecker method)
   (in module subprocess)
check_py3k_warnings() (in module test.test_support)
check_unused_args() (string.Formatter method)
check_warnings() (in module

compression() (ssl.SSL\$
CompressionError
compressobj() (in module
COMSPEC, [1]
concat() (in module operator)
concatenation
operation
Condition (class in multiprocessing.managers.SyncManager method)
Conditional
expression
conditional
expression
ConfigParser (class in C (module)
configuration
file
file, debugger
file, path
file, user
configuration information
configure() (tk.Style method)
confstr() (in module os)
confstr_names (in module
conjugate() (complex number
(decimal.Decimal method)
(numbers.Complex method)
connect() (asyncore.dispatcher method)
(ftp.lib.FTP method)
(httplib.HTTPConnection (in module sqlite3)
(multiprocessing.managers SyncManager method)
test.test_support()
checkbox() (msilib.Dialog method)
checkcache() (in module linecache)
checkfuncname() (in module bdb)
CheckList (class in Tix)
checksum
    Cyclic Redundancy Check
    MD5
    SHA
chflags() (in module os)
chgat() (curses.window method)
childerr (popen2.Popen3 attribute)
childNodes (xml.dom.Node attribute)
chmod() (in module os)
choice() (in module random)
choices (optparse.Option attribute)
choose_boundary() (in module mimetools)
chown() (in module os)
chr
    built-in function, [1]
chr() (built-in function)
chroot() (in module os)
Chunk (class in chunk)
chunk (module)
cipher
    DES
cipher() (ssl.SSLSocket method)
circle() (in module turtle)
CIRCUMFLEX (in module token)
CIRCUMFLESEX (in module token)
Clamped (class in decimal)
class
    attribute
    attribute assignment
    classic
    constructor
    definition, [1]
(smtplib.SMTP method)
    (socket.socket method)
connect_ex() (socket.socket method)
Connection (class in multiprocessing)
    (class in sqlite3)
connection (sqlite3.Cursor)
ConnectRegistry() (in module _winreg)
const (optparse.Option attribute)
    constant
    class
    constructor()
    (in module optparse)
    (in module collections)
contains() (in module operator)
content
    type
    MIME
ContentHandler (class in xml.sax.handler)
    contents (ctypes._Pointer)
ContentTooShortError
Context (class in decimal)
    context
    management
    protocol
    context
    manager, [1], [2]
context_diff() (in module difflib)
contextlib (module)
    contextmanager()
    (in module contextlib)
continue
    statement, [1], [2], [3]
CONCURRENTLOOP (opcode)
Control (class in msilib)
    (class in Tix)
control (in module cd)
control() (msilib.Dialog method)
    (select.kqueue method)
controlnames (in module cd)
instance
name
new-style
object, [1], [2], [3], [4]
old-style
statement
Class (class in symtable)
Class browser
class instance
  attribute
  attribute assignment
call
  object, [1], [2], [3]
class object
  call, [1], [2], [3]
classic class
classmethod
  built-in function
classmethod() (built-in function)
classobj() (in module new)
ClassType (in module types), [1]
clause
clean() (mailbox.Maildir method)
cleandoc() (in module inspect)
cleanup functions
Clear Breakpoint
clear() (collections.deque method)
  (cookielib.CookieJar method)
  (curses.window method)
  (dict method)
  (in module turtle), [1]
  (mailbox.Mailbox method)
  (set method)
  (threading.Event method)
  (xml.etree.ElementTree.Element method)
clear_all_breaks() (bdb.Bdb method)
controls() (ossaudiodev. method)
conversion
  arithmetic
  string, [1], [2]
ConversionError
conversions
  numeric
  convert() (email.charset.
    convert_arg_line_to_arg
    (argparse.ArgumentParser
    convert_charref() (sgml
    convert_codepoint() (sgml
    convert_entityref() (sgml
    convert_field() (string.
    convert_path() (in module
class Cookie (class in cookieli
    (module)
CookieError
CookieJar (class in cook
  cookiejar (urllib2.HTTPC
  attribute)
cookielib (module)
CookiePolicy (class in co
  Coordinated Universal T
Copy
copy
  module
copy (module)
copy() (decimal.Context
    (dict method)
    (hashlib.hash method)
    (hmac.HMAC method)
    (imaplib.IMAP4 meth
clear_all_file_breaks() (bdb.Bdb method)
clear_bpbynumber() (bdb.Bdb method)
clear_break() (bdb.Bdb method)
clear_flags() (decimal.Context method)
clear_history() (in module readline)
clear_memo() (pickle.Pickler method)
clear_session_cookies() (cookielib.CookieJar method)
clearcache() (in module linecache)
ClearData() (msilib.Record method)
clearok() (curses.window method)
clearscreen() (in module turtle)
clearstamp() (in module turtle)
clearstamps() (in module turtle)
Client() (in module multiprocessing.connection)
  client_address (BaseHTTPServer.BaseHTTPRequestHandler attribute)
clock() (in module time)
clone() (email.generator.Generator method)
  (in module turtle)
  (pipes.Template method)
clonenode() (xml.dom.minidom.Node method)
  (xml.dom.Node method)
Close() (_winreg.PyHKEY method)
close() (aifc.aifc method), [1]
  (FrameWork.Window method)
  (HTMLParser.HTMLParser method)
  (asyncore.dispatcher method)
  (bsddb.bsddbobject method)
  (bz2.BZ2File method)
  (chunk.Chunk method)
  (distutils.text_file.TextFile method)
  (dl.df method)
  (email.parser.FeedParser method)
  (file method)
  (in module copy)
  (in module findertools)
  (in module macostools)
  (in module multiprocessing)
  (in module shutil)
  (md5.md5 method)
  (pipes.Template method)
  (set method)
  (sha.sha method)
  (zlib.Compress method)
  (zlib.Decompress method)
copy2() (in module shutil)
copy_abs() (decimal.Context method)
  (decimal.Decimal method)
copy_decimal() (decimal.Decimal method)
copy_file() (in module distutils.file_util)
copy_location() (in module shutil)
copy_negate() (decimal.Decimal method)
copy_reg (module)
copy_sign() (decimal.Decimal method)
copy_tree() (in module shutil)
copybinary() (in module shutil)
copyfile() (in module shutil)
copyfileobj() (in module shutil)
copying files
copyliteral() (in module mimetools)
copypassword() (milib.FileMessage method)
copymode() (in module shutil)
copyright (built-in variable)
  (in module sys), [1]
copysign() (in module math)
copystat() (in module shutil)
copystree() (in module shutil)
copytree() (in module shutil)
coroutine
(ftplib.FTP method)
(generator method)
(hotshot.Profile method)
(httplib.HTTPConnection method)
(imaplib.IMAP4 method)
(in module anydbm)
(in module dbm)
(in module dumbdbm)
(in module fileinput)
(in module gdbm)
(in module os), [1]
(io.IOBase method)
(logging.FileHandler method)
(logging.Handler method)
(logging.handlers.MemoryHandler method)
(logging.handlers.NTEventLogHandler method)
(logging.handlers.SocketHandler method)
(logging.handlers.SysLogHandler method)
(mailbox.MH method)
(mailbox.Mailbox method)
(mailbox.Maildir method)
(mmap.mmap method)
Close() (msilib.View method)
close() (multiprocessing.Connection method)
(StringIO.StringIO method)
(multiprocessing.Queue method)
(multiprocessing.connection.Listener method)
(multiprocessing.pool.multiprocessing.Pool method)
(ossaudiodev.oss_audio_device method)
(ossaudiodev.oss_mixer_device method)
(select.epoll method)
cos() (in module cmath)
   (in module math)
cosh() (in module cmath)
   (in module math)
count() (array.array method)
   (collections.deque method)
   (in module itertools)
   (in module string)
   (list method)
   (str method)
Counter (class in collections)
countOf() (in module operator)
countTestCases() (unittest.TestCase method)
CoverageResults (class)
cPickle
   module
cPickle (module)
CPP
CPPFLAGS
cProfile (module)
CPU time
cpu_count() (in module multiprocessing)
CPython
CRC (zipfile.ZipInfo attribute)
crc32() (in module binascii)
   (in module zlib)
crc_hqx() (in module binascii)
crc32() (in module binascii)
create() (imaplib.IMAP4 method)
create_aggregate() (sqlite3.Connection method)
create_collation() (sqlite3.Connection method)
create_connection() (in module sqlite3)
create_decimal() (decimal.Context method)
create_decimal_from_float() (decimal.Context method)
create_default_context()
cmdloop() (cmd.Cmd method)
cmdqueue (cmd.Cmd attribute)
cmp
    built-in function, [1], [2], [3]
cmp() (built-in function)
    (in module filecmp)
cmp_op (in module dis)
cmp_to_key() (in module functools)
cmpfiles() (in module filecmp)
co_argcount (code object attribute)
co_cellvars (code object attribute)
co_code (code object attribute)
co_consts (code object attribute)
co_filename (code object attribute)
co_firstlineno (code object attribute)
co_flags (code object attribute)
co_freevars (code object attribute)
co_lnotab (code object attribute)
co_name (code object attribute)
co_names (code object attribute)
co_nlocals (code object attribute)
co_stacksize (code object attribute)
co_varnames (code object attribute)
CObject
    object
code
    block
code (module)
    (urllib2.HTTPError attribute)
    (xml.parsers.expat.ExpatError attribute)
crypt()
    (built-in function)
    (in module logging)
critical() (in module logging)
    (logging.Logger method)
CRNCYSTR (in module locale)
cross() (in module audio)
crypt
    module
crypt()
    (in module crypto)
crypt(3), [1], [2]
cryptography
    module
cStringIO (module)
csv
    module
ctermid() (in module os)
current_process() (in module multiprocessing)
current_thread() (in module threading)
currentframe() (in module inspect)
CTRL_BREAK_EVENT (in module signal)
CTRL_C_EVENT (in module signal)
crypt
    module
crypt()
    (in module crypto)
crypt(3), [1], [2]
cryptography
    module
cStringIO (module)
csv
    module
ctermid() (in module os)
current_process() (in module multiprocessing)
current_thread() (in module threading)
currentframe() (in module inspect)
CTRL_BREAK_EVENT (in module signal)
CTRL_C_EVENT (in module signal)
crypt
    module
crypt()
    (in module crypto)
crypt(3), [1], [2]
cryptography
    module
cStringIO (module)
csv
    module
ctermid() (in module os)
current_process() (in module multiprocessing)
current_thread() (in module threading)
currentframe() (in module inspect)
CTRL_BREAK_EVENT (in module signal)
CTRL_C_EVENT (in module signal)
crypt
    module
crypt()
    (in module crypto)
crypt(3), [1], [2]
cryptography
    module
cStringIO (module)
csv
    module
ctermid() (in module os)
current_process() (in module multiprocessing)
current_thread() (in module threading)
currentframe() (in module inspect)
CTRL_BREAK_EVENT (in module signal)
CTRL_C_EVENT (in module signal)
crypt
    module
crypt()
    (in module crypto)
crypt(3), [1], [2]
cryptography
    module
cStringIO (module)
csv
    module
ctermid() (in module os)
current_process() (in module multiprocessing)
current_thread() (in module threading)
currentframe() (in module inspect)
CTRL_BREAK_EVENT (in module signal)
CTRL_C_EVENT (in module signal)
crypt
    module
crypt()
    (in module crypto)
crypt(3), [1], [2]
cryptography
    module
cStringIO (module)
csv
    module
ctermid() (in module os)
current_process() (in module multiprocessing)
current_thread() (in module threading)
currentframe() (in module inspect)
CTRL_BREAK_EVENT (in module signal)
CTRL_C_EVENT (in module signal)
crypt
    module
crypt()
    (in module crypto)
crypt(3), [1], [2]
cryptography
    module
cStringIO (module)
csv
    module
ctermid() (in module os)
current_process() (in module multiprocessing)
current_thread() (in module threading)
currentframe() (in module inspect)
CTRL_BREAK_EVENT (in module signal)
CTRL_C_EVENT (in module signal)
crypt
    module
crypt()
    (in module crypto)
crypt(3), [1], [2]
cryptography
    module
cStringIO (module)
csv
    module
ctermid() (in module os)
current_process() (in module multiprocessing)
current_thread() (in module threading)
currentframe() (in module inspect)
CTRL_BREAK_EVENT (in module signal)
CTRL_C_EVENT (in module signal)
crypt
    module
crypt()
    (in module crypto)
crypt(3), [1], [2]
cryptography
    module
cStringIO (module)
csv
    module
ctermid() (in module os)
current_process() (in module multiprocessing)
current_thread() (in module threading)
currentframe() (in module inspect)
CTRL_BREAK_EVENT (in module signal)
CTRL_C_EVENT (in module signal)
crypt
    module
crypt()
    (in module crypto)
crypt(3), [1], [2]
cryptography
    module
cStringIO (module)
csv
    module
ctermid() (in module os)
current_process() (in module multiprocessing)
current_thread() (in module threading)
currentframe() (in module inspect)
CTRL_BREAK_EVENT (in module signal)
CTRL_C_EVENT (in module signal)
crypt
    module
crypt()
    (in module crypto)
crypt(3), [1], [2]
cryptography
    module
cStringIO (module)
csv
    module
ctermid() (in module os)
current_process() (in module multiprocessing)
current_thread() (in module threading)
currentframe() (in module inspect)
CTRL_BREAK_EVENT (in module signal)
CTRL_C_EVENT (in module signal)
crypt
    module
crypt()
    (in module crypto)
crypt(3), [1], [2]
cryptography
    module
cStringIO (module)
csv
    module
ctermid() (in module os)
current_process() (in module multiprocessing)
current_thread() (in module threading)
currentframe() (in module inspect)
CTRL_BREAK_EVENT (in module signal)
CTRL_C_EVENT (in module signal)
crypt
    module
crypt()
D_FMT (in module locale)
D_T_FMT (in module locale)
daemon (multiprocessing.Process attribute)
        (threading.Thread attribute)
dangling
        else
data
        packing binary
        tabular
        type
        type, immutable
Data (class in plistlib)
data (select.kevent attribute)
        (UserDict.IterableUserDict attribute)
        (UserList.UserList attribute)
        (UserString.MutableString attribute)
        (xml.dom.Comment attribute)
        (xml.dom.ProcessingInstruction attribute)
        (xml.dom.Text attribute)
        (xmlrpclib.Binary attribute)
data() (xml.etree.ElementTree.TreeBuilder method)
database
        Unicode
databases
DatagramHandler (class in logging.handlers)
DatagramRequestHandler (class in SocketServer)
DATASIZE (in module cd)
date (class in datetime)
date() (datetime.datetime method)
        (ntplib.NNTP method)
date_time (zipfile.ZipInfo attribute)
date_time_string()
datetime (class in datetime)
DateTime (class in xmlrpcLib)
datetime (module)
datum
day (datetime.date attribute)
  (datetime.datetime attribute)
day_abbr (in module calendar)
day_name (in module calendar)
daylight (in module time)
Daylight Saving Time
DbfilenameShelf (class in shelve)
dbhash
  module
dbhash (module)
dbm
  module, [1], [2], [3]
dbm (module)
deactivate_form() (fl.form method)
deallocation, object
debug (imaplib.IMAP4 attribute)
DEBUG (in module re)
debug (shlex.shlex attribute)
  (zipfile.ZipFile attribute)
debug() (in module doctest)
  (in module logging)
    (logging.Logger method)
    (pipes.Template method)
    (unittest.TestCase method)
    (unittest.TestSuite method)
DEBUG_COLLECTABLE (in module gc)
DEBUG_INSTANCES (in module gc)
DEBUG_LEAK (in module gc)
DEBUG_OBJECTS (in module gc)
debug_print() (distutils.ccompiler.CCompiler method)
DEBUG_SAVEALL (in module gc)
debug_src() (in module doctest)
DEBUG_STATS (in module gc)
DEBUG_UNCOLLECTABLE (in module gc)
debugger, [1], [2]
  configuration file
debugging
  CGI
  assertions
DebuggingServer (class in smtpd)
DebugRunner (class in doctest)
DebugStr() (in module MacOS)
Decimal (class in decimal)
decimal (module)
decimal literal
decimal() (in module unicodedata)
DecimalException (class in decimal)
decode
  Codecs
decode() (codecs.Codec method)
    (codecs.IncrementalDecoder method)
      (in module base64)
      (in module codecs)
      (in module mimetools)
      (in module quopri)
      (in module uu)
    (json.JSONDecoder method)
    (str method)
    (xmlrpclib.Binary method)
    (xmlrpclib.DateTime method)
decode_header() (in module email.header)
decode_params() (in module email.utils)
decode_rfc2231() (in module email.utils)
DecodedGenerator (class in email.generator)
decodestring() (in module base64)
  (in module quopri)
decomposition() (in module unicodedata)
distutils.dep_util (module)
distutils.dir_util (module)
distutils.dist (module)
distutils.emxccompiler (module)
distutils.errors (module)
distutils.extension (module)
distutils.fancy_getopt (module)
distutils.file_util (module)
distutils.filelist (module)
distutils.log (module)
distutils.msvccompiler (module)
distutils.spawn (module)
distutils.sysconfig (module)
distutils.text_file (module)
distutils.unixxcompiler (module)
distutils.util (module)
distutils.version (module)
DISTUTILS_DEBUG
dither2grey2() (in module)
dither2mono() (in module)
div() (in module operator)
divide() (decimal.Context method)
divide_int() (decimal.Context method)
  integer
  long integer
DivisionByZero (class in divmod)
  built-in function, [1], [2]
divmod() (built-in function)
  (decimal.Context method)
dl (module)
DllCanUnloadNow() (in module)
DllGetClassObject() (in module)
dllhandle (in module sys)
dngettext() (in module gettext)
do_activate() (FrameWork.ScrolledWindow method)
decompress() (bz2.BZ2Decompressor method)
    (in module bz2)
    (in module jpeg)
    (in module zlib)
    (zlib.Decompress method)

decompressobj() (in module zlib)

decorator

DEDENT (in module token)

DEDENT token, [1]

dedent() (in module textwrap)

deeepcopy() (in module copy)

def
    statement

def_prog_mode() (in module curses)

def_shell_mode() (in module curses)

default
    parameter value

default (optparse.Option attribute)
default() (cmd.Cmd method)
    (compiler.visitor.ASTVisitor method)
    (json.JSONEncoder method)

DEFAULT_BUFFER_SIZE (in module io)
default_bufsize (in module xml.dom.pulldom)
default_factory (collections.defaultdict attribute)

DEFAULT_FORMAT (in module tarfile)
default_open() (urllib2.BaseHandler method)
default_timer() (in module timeit)
DefaultContext (class in decimal)
DefaultCookiePolicy (class in cookielib)
defaultdict (class in collections)
DefaultHandler() (xml.parsers.expat.xmlparser method)
DefaultHandlerExpand()
    (xml.parsers.expat.xmlparser method)
defaults() (ConfigParser.RawConfigParser method)

(FrameWork.Window

do_char() (FrameWork./
do_clear() (bdb.Bdb method)
do_command() (curses.
do_contentclick() (Frame
do_controlhit() (FrameW

(FrameWork.Scrollec

do_dialogevent() (Frame

do_forms() (in module fl',
do_GET() (SimpleHTTPSVer.method)
do_handshake() (ssl.SS

do_HEAD() (SimpleHTTPSVer.method)
do_itemhit() (FrameW

do_POST() (CGIHTTPSVer.method)
do_postresize() (FrameW

(FrameWork.Window

do_update() (FrameW

doc_header (cmd.Cmd attribute)
DocCGIXMLRPCReque
DocXMLRPCServer
DocFileSuite() (modul
doCleanups() (unittest.
docmd() (smtplib.SMTP

docstring, [1]
doctest.Doctest attr
docstrings, [1]

DocTest (class in doctes doctest (module)
DocTestFailure
defaultTestLoader (in module unittest)
defaultTestResult() (unittest.TestCase method)
defects (email.message.Message attribute)
define_macro() (distutils.ccompiler.CCompiler method)
definition
class, [1]
function, [1]
defpath (in module os)
degrees() (in module math)
    (in module turtle)
del
    statement, [1], [2], [3]
del_param() (email.message.Message method)
delattr() (built-in function)
delay() (in module turtle)
delay_output() (in module curses)
delayload (cookielib.FileCookieJar attribute)
delch() (curses.window method)
dele() (poplib.POP3 method)
delete() (ftplib.FTP method)
    (imaplib.IMAP4 method)
    (ttk.Treeview method)
DELETE_ATTR (opcode)
DELETE_FAST (opcode)
DELETE_GLOBAL (opcode)
DELETE_NAME (opcode)
DELETE_SLICE+0 (opcode)
DELETE_SLICE+1 (opcode)
DELETE_SLICE+2 (opcode)
DELETE_SLICE+3 (opcode)
DELETE_SUBSCR (opcode)
deleteacl() (imaplib.IMAP4 method)
deletefilehandler() (Tkinter.Widget.tk method)
deletefolder() (mhlib.MH method)
DeleteKey() (in module _winreg)
DocTestFinder (class in doctest)
DocTestParser (class in doctest)
DocTestRunner (class in doctest)
DocTestSuite() (in module doctest)
doctype() (xml.etree.ElementTree method)
documentation
generation
    online
documentation string
documentation strings, [1]
documentElement (xml.dom.Element method)
DocXMLRPCRequestHandler (class in DocXMLRPCServer)
DocXMLRPCServer (class in DocXMLRPCServer)
    (module)
domain_initial_dot (cookielib.Cookie attribute)
domain_return_ok() (cookielib.CookiePolicy method)
domain_specified (cookielib.Cookie attribute)
DomainLiberal (cookielib.DefaultCookiePolicy attribute)
DomainRFC2965Match (cookielib.DefaultCookiePolicy attribute)
DomainStrict (cookielib.DefaultCookiePolicy attribute)
DomainStrictNoDots (cookielib.DefaultCookiePolicy attribute)
DomainStrictNonDomain (cookielib.DefaultCookiePolicy attribute)
DOMEventStream (class in xml.dom.pulldom)
DOMException
    DomstringSizeErr
done() (in module turtle)
    (xdrlib.Unpacker met
devnull (in module os)
dgettext() (in module gettext)
Dialect (class in csv)
dialect (csv.csvreader attribute)
    (csv.csvwriter attribute)
Dialog (class in msilib)
DialogWindow() (in module FrameWork)
dict (2to3 fixer)
    (built-in class)
dict()
(multiprocessing.managers.SyncManager method)
dictConfig() (in module logging.config)
dictionary
    display
        object, [1], [2], [3], [4], [5], [6], [7]
        type, operations on
dictionary view
DictionaryType (in module types), [1]
DictMixin (class in UserDict)
DictProxyType (in module types)
DictReader (class in csv)
DictType (in module types), [1]
DictWriter (class in csv)
diff_files (filecmp.dircmp attribute)
Differ (class in difflib), [1]
difference() (set method)
difference_update() (set method)
difflib (module)
digest() (hashlib.hash method)
    (hmac.HMAC method)
    (md5.md5 method)
    (sha.sha method)
digest_size (in module md5)
    (in module sha)
digit() (in module unicodedata)
digits (in module string)

dump_address_pair() (in module rfc822)
dump_stats() (profile.Prof method)
    (pstats.Stats method)
dumps() (in module json)
    (in module marshal)
    (in module pickle)
    (in module xmlrpclib)
dup() (in module os)
    (posixfile.posixfile method)
dup2() (in module os)
    (posixfile.posixfile method)
DUP_TOP (opcode)
DUP_TOPX (opcode)
DuplicateSectionError
dwFlags (subprocess.STARTUPINFO)
DynLoadSuffixImporter
dir() (built-in function)
   (ftplib.FTP method)
dircache (module)
dircmp (class in filecmp)
directory
   changing
   creating
   deleting, [1]
site-packages
site-python
traversal
walking
Directory (class in msilib)
Index – E

e (in module cmath)
  (in module math)
E2BIG (in module errno)
EACCES (in module errno)
EADDRINUSE (in module errno)
EADDRNOTAVAIL (in module errno)
EADV (in module errno)
EAFNOSUPPORT (in module errno)
\textbf{EAFP}
EAGAIN (in module errno)
EALREADY (in module errno)
east_asian_width() (in module unicodedata)
EasyDialogs (module)
EBADE (in module errno)
EBADF (in module errno)
EBADFD (in module errno)
EBADMSG (in module errno)
EBADR (in module errno)
EBADRQC (in module errno)
EBADSLT (in module errno)
EBCDIC
EBFONT (in module errno)
EBUSY (in module errno)
ECHILD (in module errno)
echo() (in module curses)
echochar() (curses.window method)
ECHNRG (in module errno)
ECOMM (in module errno)
ECONNABORTED (in module errno)
ECONNREFUSED (in module errno)
ECONNRESET (in module errno)
EDEADLK (in module errno)
EDEADLOCK (in module errno)
EDESTADDRREQ (in module errno)
edit() (curses.textpad.Textbox method)
EDOM (in module errno)
EPFNOSUPPF
epilogue (email.message.Message attribute)
EPIPE (in module errno)
epoll() (in module select)
EPROTO (in module errno)
EPROTONOSUPPORT (in module errno)
EPROTOTYP
eq() (in module math)
EQUAL (in module token)
ERA (in module locale)
ERA_D_FMT (in module locale)
ERA_D_T_FMT (in module locale)
ERA_T_FMT (in module locale)
ERANGE (in module errno)
erase() (curses.window method)
erasechar() (in module curses)
EREADCHG (in module errno)
EREMOTE (in module errno)
EREMOTEIO
EREMOTEIO (in module errno)
EERESTART (in module errno)
erf() (in module math)
erfc() (in module math)
EROFS (in module errno)
ERR (in module error)
errcode (ctypes._FuncPtr attribute)
ererrmsg (xmlrpclib.ProtocolError attribute)
errno (module, [1]
Error, [1], [2], [12]
error, [1], [2], [12], [13], [14]
EDOTDOT (in module errno) [22], [23], [24]
EDQUOT (in module errno)
EEXIST (in module errno)
EFAULT (in module errno)
EFBIG (in module errno)
effective() (in module bdb)
ehlo() (smtplib.SMTP method)
ehlo_or_helo_if_needed() (smtplib.SMTP method)
EHOSTDOWN (in module errno)
EHOSTUNREACH (in module errno)
EIDRM (in module errno)
EILSEQ (in module errno)
EINPROGRESS (in module errno)
EINTR (in module errno)
EINVAL (in module errno)
EIO (in module errno)
EISCONN (in module errno)
EISDIR (in module errno)
EISNAM (in module errno)
EL2HLT (in module errno)
EL2NSYNC (in module errno)
EL3HLT (in module errno)
EL3RST (in module errno)
Element (class in xml.etree.ElementTree)
element_create() (ttk.Style method)
element_names() (ttk.Style method)
element_options() (ttk.Style method)
ElementDeclHandler() (xml.parsers.expat.xmlparser method)
elements() (collections.Counter method)
ElementTree (class in xml.etree.ElementTree)
ELIBACC (in module errno)
ELIBBAD (in module errno)
ELIBEXEC (in module errno)
ELIBMAX (in module errno)
ELIBSCN (in module errno)
elif
    keyword
Ellinghouse, Lance
Ellipsis

- object
- Ellipsis (built-in variable)
- ELLIPSIS (in module doctest)
- EllipsisType (in module types)
- ELNRNG (in module errno)
- ELOOP (in module errno)
- else
  - dangling keyword, [1], [2], [3], [4]
- email (module)
- email.charset (module)
- email.encoders (module)
- email.errors (module)
- email.generator (module)
- email.header (module)
- email.iterators (module)
- email.mime (module)
- email.parser (module)
- email.utils (module)
- EMFILE (in module errno)
- emit() (logging.FileHandler method)
  - (logging.Handler method)
  - (logging.NullHandler method)
  - (logging.StreamHandler method)
  - (logging.handlers.BufferingHandler method)
  - (logging.handlers.DatagramHandler method)
  - (logging.handlers.HTTPHandler method)
  - (logging.handlers.NTEventLogHandler method)
  - (logging.handlers.RotatingFileHandler method)
  - (logging.handlers.SMTPHandler method)
  - (logging.handlers.SocketHandler method)
  - (logging.handlers.SysLogHandler method)
  - (logging.handlers.TimedRotatingFileHandler method)
  - (logging.handlers.WatchedFileHandler method)
- EMLINK (in module errno)

Errors

- logging
  - errors
    - (file attribute)
    - (io.TextIOBase attribute)
    - (unittest.TestCase attribute)
    - ErrorString() (xml.parsers.expat ErrorString) escape (shlex.escape) escape (shlex.escape_sequence) escape() (in module) escape() (in module)
    - (in module)
  - escapechar (csv.escapechar)
  - escapedquotes (shlex.escapedquotes)
  - ESHUTDOWN (in module)
  - ESOCKTNOSUPPORT (in module)
  - ESPIPE (in module)
  - ESRCH (in module)
  - ESRMNT (in module)
  - ESTALE (in module)
  - ESTRPIPE (in module)
  - ETIMEDOUT (in module)
  - Etiny() (decimal.Context method)
  - ETOOMANYREFS (in module)
  - ETOP() (decimal.Context method)
  - ETXTBSY (in module)
  - EUCLEAN (in module)
  - EUNATCH (in module)
  - EUSERS (in module)
eval built-in function
  - eval() (built-in function)
evaluation
  - Event (class in multiprocess)
Empty
empty
  list
tuple, [1]
empty()
(multiprocessing.multiprocessing.queues.SimpleQueue method)
  (Queue.Queue method)
  (multiprocessing.Queue method)
  (sched.scheduler method)
EMPTY_NAMESPACE (in module xml.dom)
emptyline() (cmd.Cmd method)
EMSGSIZE (in module errno)
EMULTIHOP (in module errno)
enable() (bdb.Breakpoint method)
  (in module cgitb)
  (in module gc)
  (profile.Profile method)
enable_callback_tracebacks() (in module sqlite3)
enable_interspersed_args() (optparse.OptionParser method)
enable_load_extension() (sqlite3.Connection method)
enable_traversal() (ttk.Notebook method)
ENABLE_USER_SITE (in module site)
EnableReflectionKey() (in module _winreg)
ENAMETOOLONG (in module errno)
ENAVAIL (in module errno)
enclose() (curses.window method)
encode
  Codecs
encode() (codecs.Codec method)
  (codecs.IncrementalEncoder method)
  (email.header.Header method)
  (in module base64)
  (in module codecs)
  (in module mimetools)
  (in module quopri)
ENOLCK (in module errno)
ENOLINK (in module errno)
ENOMEM (in module errno)
ENOMEM (in module errno)
ENOMS (in module errno)
ENOPROTOOPT (in module errno)
ENOSSPC (in module errno)
ENOSR (in module errno)
ENOSTR (in module errno)
ENOSYS (in module errno)
ENOTBLK (in module errno)
ENOTCONN (in module errno)
ENOTDIR (in module errno)
ENOTEMPTY (in module errno)
ENOTNAM (in module errno)
ENOTSOCK (in module errno)
ENOTTY (in module errno)
ENOTUNIQ (in module errno)
enurepip (module)
enter() (sched.scheduler method)
enterabs() (sched.scheduler method)
entities (xml.dom.DocumentType attribute)
EntityDeclHandler() (xml.parsers.expat.xmlparser method)
entitydefs (in module htmlentitydefs)
EntityResolver (class in xml.sax.handler)
Enum (class in aetypes)
enumerate() (built-in function)
    (in module fm)
    (in module threading)
EnumKey() (in module _winreg)
enumsubst() (in module aetools)
EnumValue() (in module _winreg)
environ (in module os)
    (in module posix)

(argparse.ArgumentParser method)
(in module sys)
(in module thread)
exitfunc (2to3 fixer)
exitonclick() (in module sys)
expm1() (in module math)
expovariate() (in module random)
expr() (in module parser)
Expression
Expression_error
ExpressionError
expect() (telnetlib.Telnet method)
expectedFailure() (in module unittest)
expectedFailures (unittest.TestResult attribute)
expires (cookielib.Cookie attribute)
expm1() (in module math)
expovariate() (in module random)
expression, [1]
Condition:
conditiona
generator
lambda, [1]
list, [1], [2]
statement
environment

environment variable

%PATH%

<protocol>_proxy

APPDATA

AUDIODEV

BROWSER, [1]

CC

CFLAGS, [1], [2]

COLUMNS, [1]

COMSPEC, [1]

CPP

CPPFLAGS

DISTUTILS_DEBUG

HOME, [1], [2], [3], [4], [5]

HOMEDRIVE, [1]

HOMEPATH, [1]

IDLESTARTUP

KDEDIR

LANG, [1], [2], [3], [4]

LANGUAGE, [1]

LC_ALL, [1]

LC_MESSAGES, [1]

LDCXXSHAREd

LDLIBS

LINES, [1], [2]

LNAME

LOGNAME, [1]

MIXERDEV

PAGER, [1]

PATH, [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18]

PLAT

POSIXLY_CORRECT

yield

expunge() (imaplib.IMAP4 method)

extend() (array.array method)

(collectio

(xml.etree.

extend_path() extended slicing

extended print

extended slice

assignment operation

extended slicing

EXTENDED_ ExtendedCon

extendleft() (c

extension

module

Extension (class

extensions_map

(SimpleHTTP

attribute)

External Data

external_attr (z

External Clash

ExternalEntity

(xml.parsers.expat.

extra (zipfile.Z

extract() (tarfi

(zipfile.Zip

extract_cookies()

extract_stack()

extract_tb() (i
PYTHON*
PYTHONCASEOK, [1]
PYTHONDEBUG, [1]
PYTHONDOCS
PYTHONDONTWRITEBYTECODE, [1, 2, 3, 4]
PYTHONDUMPREFS, [1]
PYTHONEXECUTABLE
PYTHONHASHSEED, [1, 2, 3]
PYTHONHOME, [1, 2, 3, 4, 5, 6, 7, 8, 9, [10], [11]
PYTHONHTTPSVERIFY
PYTHONINSPECT, [1, 2]
PYTHONIOENCODING, [1]
PYTHONMALLOCSTATS
PYTHONNOUSERSITE, [1, 2, 3]
PYTHONOPTIMIZE, [1]
PYTHONPATH, [1, 2, 3, 4, 5, 6, 7, 8, 9, [10], [11], [12], [13], [14], [15], [16], [17]
PYTHONSTARTUP, [1, 2, 3, 4, 5, 6, 7]
PYTHONTHREADDEBUG
PYTHONUNBUFFERED, [1]
PYTHONUSERBASE, [1, 2, 3]
PYTHONVERBOSE, [1]
PYTHONWARNINGS, [1, 2, 3]
PYTHON2K, [1, 2, 3]
PYTHON_DOM
SystemRoot
TCL_LIBRARY
TEMP
TERM, [1]
TIX_LIBRARY
TK_LIBRARY
TMP, [1]
TMPDIR, [1]
TZ, [1], [2], [3], [4]
USER
USERNAME
USERPROFILE, [1]
USER_BASE
Wimp$ScrapDir
exec_prefix, [1], [2]
ftp_proxy
http_proxy, [1], [2], [3]
no_proxy, [1]
prefix, [1], [2], [3]

environment variables

deleting

setting

EnvironmentError

EnvironmentVarGuard (class in test.test_support)
ENXIO (in module errno)
eof (shlex.shlex attribute)
EOFError

(built-in exception)
EOPNOTSUPP (in module errno)
EOVERFLOW (in module errno)
EPERM (in module errno)
Index – F

f_back (frame attribute)
F_BAVAIL (in module statvfs)
F_BFREE (in module statvfs)
F_BLOCKS (in module statvfs)
F_BSIZE (in module statvfs)
f_builtins (frame attribute)
f_code (frame attribute)
f_exc_traceback (frame attribute)
f_exc_type (frame attribute)
f_exc_value (frame attribute)
F_FAVAIL (in module statvfs)
F_FFREE (in module statvfs)
F_FILES (in module statvfs)
F_FLAG (in module statvfs)
F_FRSIZE (in module statvfs)
f_globals (frame attribute)
f_lasti (frame attribute)
f_lineno (frame attribute)
f_locals (frame attribute)
F_NAME_MAX (in module statvfs)
F_OK (in module os)
f_restricted (frame attribute)
f_trace (frame attribute)
fabs() (in module math)
factorial() (in module math)
fail() (unittest.TestCase method)
failfast (unittest.TestResult attribute)
failureException (unittest.TestCase attribute)
failures (unittest.TestResult attribute)
False, [1], [2]
false
False (Built-in object)
(built-in variable)
family (socket.socket attribute)
fancy_getopt() (in module

flatten() (email.generator.Generator method)
flattening
objects
float
built-in function, [1], [2], [3
float (built-in class)
float_info (in module sys)
float_repr_style (in module sys)
floating point
literals
number
object, [1], [2]
floating point literal
FloatingPointError, [1]
FloatType (in module types)
in modules types
flock() (in module fcntl)
floor division
floor() (in module math), [1]
floordiv() (in module operator
floppy (module)
flush() (bz2.BZ2Compressor
(file method)
(formatter.writer method)
(io.BufferedWriter method)
(io.IOBase method)
(logging.Handler method)
(logging.StreamHandler method)
(logging.handlers.BufferingHandler method)
(logging.handlers.MemoryHandler method)
(xml.sax.xmlreader.IncrementalParser
method)
FeedParser (class in email.parser)
fetch() (imaplib.IMAP4 method)
Fetch() (msilib.View method)
fetchall() (sqlite3.Cursor method)
fetchmany() (sqlite3.Cursor method)
fetched() (sqlite3.Cursor method)
fflags (select kevent attribute)
field_size_limit() (in module csv)
fieldnames (csv.csvreader attribute)
fields (uuid.UUID attribute)
fifo (class in asynchat)
file
.ini
.pdbrc
.pythonrc.py
built-in function
byte-code, [1], [2]
configuration
copying
descriptor
large files
mime.types
object, [1], [2], [3], [4]
path configuration
plist
temporary
user configuration
file (pyclbr.Class attribute)
(pyclbr.Function attribute)
file ...
compileall command line option
file control
UNIX

Form (class in Tix)

format
str
format (memoryview attribute
(struct.Struct attribute)
format() (built-in function)
in module locale)
(logging.Formatter method)
(logging.Handler method)
(pprint.PrettyPrinter method)
(str method)
(string.Formatter method)
format_exc() (in module traceback)
format_exception() (in module traceback)
format_exception_only() (in module traceback)
format_field() (string.Formatter method)
format_help()
(argparse.ArgumentParser method)
format_list() (in module traceback)
format_stack() (in module traceback)
format_stack_entry() (bdb.Bdb method)
format_string() (in module locale)
format_tb() (in module traceback)
format_usage()
(argparse.ArgumentParser method)
formataddr() (in module email)
formatargspec() (in module inspect)
formatargvalues() (in module inspect)
formatdate() (in module email)
FormatException
FormatException() (in module ctypes)
formatException() (logging.Formatter method)
formatmonth()
(multiprocessing.Connection method)
(ossaudiodev.oss_audio_device method)
(ossaudiodev.oss_mixer_device method)
(select.epoll method)
(select.kqueue method)
(socket.socket method)
(telnetlib.Telnet method)
fileopen() (in module posixfile)
FileSelectBox (class in Tix)
FileType (class in argparse)
    (in module types), [1]
FileWrapper (class in wsgiref.util)
fill() (in module textwrap)
    (in module turtle)
        (textwrap.TextWrapper method)
fillcolor() (in module turtle)
filter (2to3 fixer)
Filter (class in logging)
filter (select.kevent attribute)
filter() (built-in function)
    (in module curses)
    (in module fnmatch)
    (in module future_builtins)
(logging.Filter method)
(logging.Handler method)
(logging.Logger method)
filterwarnings() (in module warnings)
finalization, of objects
finalize_options()
    (distutils.cmd.Command method)
finally
    keyword, [1], [2], [3], [4]
find() (doctest.DocTestFinder method)
    (in module gettext)
(in module string)
(mmap.mmap method)
(str method)
(xml.etree.ElementTree.Element method)
(xml.etree.ElementTree.ElementTree method)
find_first() (fl.form method)
find_global() (pickle protocol)
find_last() (fl.form method)
find_library() (in module ctypes.util)
find_library_file()
(distutils.ccompiler.CCompiler method)
find_loader() (in module pkgutil)
find_longest_match()
(difflib.SequenceMatcher method)
find_module
  finder
find_module() (imp.NullImporter method)
  (in module imp)
    (zipimport.zipimporter method)
find_msvcrt() (in module ctypes.util)
find_user_password()
(urllib2.HTTPPasswordMgr method)
findall() (in module re)
  (re.RegexObject method)
(xml.etree.ElementTree.Element method)
(xml.etree.ElementTree.ElementTree method)
findCaller() (logging.Logger method)
findex, [1]
    find_module
findertools (module)
findfactor() (in module audioop)
findfile() (in module test.test_support)
findfit() (in module audioop)
method)
  (datetime.datetime class method)
fromstring() (array.array method)
  (in module xml.etree.ElementTree)
fromstringlist() (in module xml.etree.ElementTree)
fromtarfile() (tarfile.TarInfo method)
fromtimestamp() (datetime.date method)
  (datetime.datetime class method)
fromunicode() (array.array method)
fromutc() (datetime.tzinfo method)
frozenset
  object, [1]
frozenset (built-in class)
fstat() (in module os)
fstatvfs() (in module os)
fsum() (in module math)
fsync() (in module os)
ftp
  ftplib (standard module)
  protocol, [1]
ftp (class in ftplib)
ftp_open() (urllib2.FTPHandler method)
ftp_proxy
FTP_TLS (class in ftplib)
FTPHandler (class in urllib2)
ftplib (module)
ftpmirror.py
ftruncate() (in module os)
Full
full() (multiprocessing.Queue
  (Queue.Queue method)
func (functools.partial attribute)
func_closure (function attribute)
func_code (function attribute)
  (function object attribute)
findfont() (in module fm)
finditer() (in module re)
    (re.RegexObject method)
findlabels() (in module dis)
findlinestarts() (in module dis)
findmatch() (in module mailcap)
findmax() (in module audioop)
findtext()
    (xml.etree.ElementTree.Element method)
    (xml.etree.ElementTree.ElementTree method)
finish()
    (SocketServer.BaseRequestHandler method)
finish_request()
    (SocketServer.BaseServer method)
first() (asynchat.fifo method)
    (bsddb.bsddbobject method)
    (dbhash.dbhash method)
firstChild (xml.dom.Node attribute)
firstkey() (in module gdbm)
firstweekday() (in module calendar)
fix() (in module fpformat)
fix_missing_locations() (in module ast)
fix_sentence_endings
    (textwrap.TextWrapper attribute)
FL (module)
fl (module)
flag_bits (zipfile.ZipInfo attribute)
flags (in module sys)
    (re.RegexObject attribute)
    (select.kevent attribute)
flags() (posixfile.posixfile method)
flash() (in module curses)

func_defaults (function attribute)
func_dict (function attribute)
func_doc (function attribute)
func_globals (function attribute)
func_name (function attribute)
funcattrs (2to3 fixer)

function
    anonymous
    argument
    call, [1], [2]
    call, user-defined
    definition, [1]
    generator, [1]
    name
    object, [1], [2], [3], [4], [5]
    user-defined
Function (class in symtable)
function() (in module new)
FunctionTestCase (class in unittest)
FunctionType (in module types)
functools (module)
funny_files (filecmp.dircmp at
future
    statement
future (2to3 fixer)
future_builtins (module)
FutureWarning
Index – G

G.722
gaierro
 gamma() (in module math)
 gammavariate() (in module random)
 garbage (in module gc)
 garbage collection, [1]
 gather() (curses.textpad.Textbox method)
 gauss() (in module random)
 gc (module)
 gcd() (in module fractions)
 gdbm
   module, [1], [2]
 gdbm (module)
 ge() (in module operator)
 gen_lib_options() (in module distutils.ccompiler)
 gen_preprocess_options() (in module distutils.ccompiler)
 gen_uuid() (in module msilib)
 generate_help() (distutils.fancy_getopt.FancyGetopt
  method)
 generate_tokens() (in module tokenize)
 generator, [1]
   expression
     function, [1], [2]
     iterator, [1]
     object, [1], [2]
 Generator (class in email.generator)
 generator expression, [1]
 GeneratorExit
   exception
 GeneratorType (in module types)
 generic
   special attribute
 generic_visit() (ast.NodeVisitor method)
 genops() (in module pickletools)
 gensuitemodule (module)
 GetErrorString() (in module MacOS)
 geteuid() (in
 getEvent()
 (xml.dom.pul
 method)
 getEventCa
 (logging.har
 method)
 getEventType
 (logging.har
 method)
 getException
 method)
 getFeature
 (xml.sax.xm
 method)
 GetFieldCount
 getfile() (in r
 getfirst() (cg
 getfirstmatch
 (rfc822.Mes
 getfloat()
 (ConfigPars
 method)
 getfmts()
 (ossaudiode
 method)
 getfqdn() (in
 getframeinfo
 getframerate
 (sunau.A
 (wave.W
 getfullname
 getgid() (in r
get_python_lib() (in module distutils.sysconfig)
get_python_version() (in module sysconfig)
get_resrc() (ossaudiodev.oss_mixer_device method)
get_referents() (in module gc)
get_referrers() (in module gc)
get_request() (SocketServer.BaseServer method)
get_rgbmode() (in module fl)
get_scheme() (wsgiref.handlers.BaseHandler method)
get_scheme_names() (in module sysconfig)
get_selector() (urllib2.Request method)
get_sequences() (mailbox.MH method)
    (mailbox.MHMessage method)
get_server() (multiprocessing.managers.BaseManager method)
get_server_certificate() (in module ssl)
gt_socket() (telnetlib.Telnet method)
gt_source() (zipimport.zipimporter method)
gt_special_folder_path() (built-in function)
gt_stack() (bdb.Bdb method)
gt_starttag_text() (HTMLParser.HTMLParser method)
    (sgmllib.SGMLParser method)
gt_stderr() (wsgiref.handlers.BaseHandler method)
    (wsgiref.simple_server.WSGIRequestHandler method)
gt_stdin() (wsgiref.handlers.BaseHandler method)
    (wsgiref.simple_server.WSGIRequestHandler method)
gt_string() (mailbox.Mailbox method)
gt_subdir() (mailbox.MaildirMessage method)
gt_suffixes() (in module imp)
gt_symbols() (symtable.SymbolTable method)
gt_terminator() (asyncchat.async_chat method)
gt_threshold() (in module gc)
gt_token() (shlex.shlex method)
gt_type() (symtable.SymbolTable method)
    (urllib2.Request method)
gt_unixfrom() (email.message.Message method)
gt_usage() (optparse.OptionParser method)
gt_value() (string.Formatter method)
gt_version() (optparse.OptionParser method)
gt_visible() (mailbox.BabylMessage method)
- getenv() (in module os)
- getErrorHandle() (xml.sax.xmlreader.XMLReader method)

- glob() (built-in function)
- globals() (built-in function)
- glob() (doctest method)
- gname (tarfile.TarInfo attribute)
- GNOME
- GNU_FORMAT (in module tarfile)
- gnu_getopt() (in module getopt)
- got (doctest method)
- goto() (in module turtle)

- Grammar

- Graphical User Interface

- GREATER (in module token)
- GREATEREQUAL (in module token)

- Greenwich Mean Time

- grey22grey() (in module imageop)
- grey2grey2() (in module imageop)
- grey2grey4() (in module imageop)
- grey2mono() (in module imageop)
- grey42grey() (in module imageop)
- group() (nnn method)
- groupby() (in module itertools)
- groupdict() (re.MatchObject method)
- groupindex (re.RegexObject attribute)
- groups (re.RegexObject attribute)
- groups() (re method)
- grp (module operator)
- gt() (in module operator)

- guess_all_extensions() (in module mimetypes)
- guess_extension() (in module mimetypes)
- guess_scheme() (in module wsgiref.util)
guess_type()
  (mimetypes)
GUI
gzip (module)
GzipFile (class in gzip)
Index – H

halfdelay() (in module curses)
handle an exception
handle()
(BaseHTTPServer.BaseHTTPRequestHandler method)
    (SocketServer.BaseRequestHandler method)
    (logging.Handler method)
    (logging.Logger method)
    (logging.NullHandler method)
    (wsgiref.simple_server.WSGIRequestHandler method)
handle_accept() (asyncore.dispatcher method)
handle_charref() (HTMLParser.HTMLParser method)
    (sgmllib.SGMLParser method)
handle_close() (asyncore.dispatcher method)
handle_comment() (HTMLParser.HTMLParser method)
    (sgmllib.SGMLParser method)
handle_connect() (asyncore.dispatcher method)
handle_data() (HTMLParser.HTMLParser method)
    (sgmllib.SGMLParser method)
handle_decl() (HTMLParser.HTMLParser method)
    (sgmllib.SGMLParser method)
handle_endtag() (HTMLParser.HTMLParser method)
    (sgmllib.SGMLParser method)
handle_entityref() (HTMLParser.HTMLParser method)
    (sgmllib.SGMLParser method)
handle_error() (asyncore.dispatcher method)
    (SocketServer.BaseServer method)
handle_expt() (asyncore.dispatcher method)
handle_image() (httplib.HTMLParser method)
handle_one_request()
(BaseHTTPServer.BaseHTTPRequestHandler method)

hex (uuid.UUID attribute)
hex() (built-in function)
    (float method)
    (in module future_builtins)
hexadecimal literals
hexadecimal literals
hexbin() (in module binhex)
hexdigest() (hashlib method)
    (hmac.HMAC method)
    (md5.md5 method)
    (sha.sha method)
hexdigits (in module string)
hexlify() (in module binascii)
hexversion (in module sys)
hidden() (curses method)
    (ttk.Notebook method)
hide_cookie2 (cookielib.CookiePolicy attribute)
hide_form() (fl.form method)
hideturtle() (in module turtle)
hierarchy
    (type
    HierarchyRequestErr)
HIGHEST_PROTOCOL (in module pickle)
HKEY_CLASSES_ROOT (in module _winreg)
HKEY_CURRENT_CONFIG (in module _winreg)
HKEY_CURRENT_USER (in module _winreg)
HKEY_DYN_DATA (in module _winreg)
has_nonstandard_attr() (cookielib.Cookie method)
HAS_NPN (in module ssl)
has_option() (ConfigParser.RawConfigParser method)
    (optparse.OptionParser method)
has_section() (ConfigParser.RawConfigParser method)
HAS_SNI (in module ssl)
hasattr() (built-in function)
hasAttribute() (xml.dom.Element method)
hasAttributeNS() (xml.dom.Element method)
hasAttributes() (xml.dom.Node method)
hasChildNodes() (xml.dom.Node method)
hascompare (in module dis)
hasconst (in module dis)
hasFeature() (xml.dom.DOMImplementation method)
hasfree (in module dis)
hash
    built-in function, [1], [2]
hash character
hash() (built-in function)
hash.block_size (in module hashlib)
hash.digest_size (in module hashlib)
hashable, [1]
    Hashable (class in collections)
hashlib (module)
hashlib.algorithms (in module hashlib)
hashopen() (in module bsddb)
hasjabs (in module dis)
hasjrel (in module dis)
haslocal (in module dis)
hasname (in module dis)
HAVE_ARGUMENT (opcode)
have_unicode (in module test.test_support)
head() (nntplib.NNTP method)
Header (class in email.header)
header_encode() (email.charset.Charset method)
header_encoding (email.charset.Charset attribute)
difflib)
htmlentitydefs (in htmllib)
    module
htmllib (module)
HTMLParseError
    HTMLParser (class in HTMLParser (module))
htnl() (in module html)
htons() (in module http)
HTTP
    httplib (standard module, protocol, [1], [2])
http_error_301() (urllib2.HTTPRedirectHandler method)
http_error_302() (urllib2.HTTPRedirectHandler method)
http_error_303() (urllib2.HTTPRedirectHandler method)
http_error_307() (urllib2.HTTPRedirectHandler method)
http_error_401() (urllib2.HTTPBasicAuthHandler method)
    (urllib2.HTTPDigestAuthHandler method)
http_error_407() (urllib2.ProxyBasicAuthHandler method)
    (urllib2.ProxyBasicAuthHandler method)
http_error_auth_reqed()
header_items() (urllib2.Request method)
header_offset (zipfile.ZipInfo attribute)
HeaderError
HeaderParseError
headers
  MIME, [1]
headers
(BaseHTTPServer.BaseHTTPRequestHandler attribute)
Headers (class in wsgiref.headers)
headers (rfc822.Message attribute)
  (xmlrpclib.ProtocolError attribute)
heading() (in module turtle)
  (ttk.Treeview method)
heapify() (in module heapq)
heapmin() (in module msvcrt)
heappop() (in module heapq)
heappush() (in module heapq)
heappushpop() (in module heapq)
heapq (module)
heapreplace() (in module heapq)
heclo() (smtplib.SMTP method)
help
  built-in function
    online
help (optparse.Option attribute)
help() (built-in function)
  (nntplib.NNTP method)
heror
hex
  built-in function
  (urllib2.AbstractError method)
    (urllib2.AbstractBaseHandler method)
http_error_default()
  (urllib2.BaseHandler method)
http_error_nnn() (urllib2.BaseHandler method)
http_open() (urllib2 method)
HTTP_PORT (in http_proxy, [1], [:]
http_response() (urllib2.HTTPError method)
http_version
  (wsgiref.handler:attribute)
HTTPBasicAuthHandler
urlib2)
HTTPConnection
urlib2)
HTTPCookieProcessor
urlib2)
httpd
HTTPDefaultErrorHandler
urlib2)
HTTPDigestAuthHandler
urlib2)
HTTPError
HTTPErrorProcessor
HTTPException
HTTPHandler (class in urllib.handlers)
httplib (module)
HTTPMessage (HTTPPasswordManager)
HTTPPasswordManager (class in urllib2)
HTTPRedirectHandler (class in urllib2)
HTTPResponse
https_open() (url method)
HTTPS_PORT (in module httplib)
https_response() (urllib2.HTTPError method)
HTTPSConnection
HTTPSConnection
HTTPSHandler (class in urllib2)
hypertext
hypot() (in module math)
Index – I

I (in module re)
I/O control
  POSIX
  UNIX
    buffering, [1], [2]
    tty
iadd() (in module operator)
iand() (in module operator)
IC (class in ic)
ic (module)
icglue
  module
iconcat() (in module operator)
icopen (module)
id
  built-in function
id() (built-in function)
    (unittest.TestCase method)
idcok() (curses.window method)
ident (in module cd)
  (select.kevent attribute)
  (threading.Thread attribute)
idchars (cmd.Cmd attribute)
identifier, [1]
identify() (ttk.Notebook method)
  (ttk.Treeview method)
  (ttk.Widget method)
identify_column() (ttk.Treeview method)
identify_element() (ttk.Treeview method)
identify_region() (ttk.Treeview method)
identify_row() (ttk.Treeview method)
identity
  test
identity of an object

internalSubset (xml.dom attribute)
Internet
Internet Config
interpolation, string (%)
InterpolationDepthError
InterpolationError
InterpolationMissingOpti
InterpolationSyntaxError
interpreted
interpreter
interpreter lock
interpreter prompts
interrupt() (sqlite3.Connection method)
interrupt_main() (in module thread)
intersection() (set method)
intersection_update() (set method)
IntlText (class in aetypes)
IntlWritingCode (class in aetypes)
intro (cmd.Cmd attribute)
IntType (in module types)
  (in modules types)
InuseAttributeErr
inv() (in module operator)
InvalidAccessErr
InvalidCharacterErr
InvalidModificationErr
InvalidOperation (class in decimal)
InvalidStateErr
InvalidURL
inversion
invert() (in module operator)
invocation
io (module)
IOBase (class in io)
ioctl() (in module fcntl)
idioms (2to3 fixer)

idiv() (in module operator)

IDLE, [1]
idle() (FrameWork.Application method)

IDLESTARTUP

idlok() (curses.window method)

IEEE-754

if

    statement, [1]

ifilter() (in module itertools)

ifilterfalse() (in module itertools)

ifloordiv() (in module operator)

iglob() (in module glob)

ignorableWhitespace()

(xml.sax.handler.ContentHandler method)

ignore_errors() (in module codecs)

IGNORE_EXCEPTION_DETAIL (in module doctest)

ignore_patterns() (in module shutil)

IGNORECASE (in module re)

ihave() (ntplib.NNTP method)

ihooks

    module

ilshift() (in module operator)

im_class (method attribute), [1]

im_func (method attribute), [1], [2], [3]

im_self (method attribute), [1], [2]

imag (numbers.Complex attribute)

imageop (module)

imaginary literal

imap() (in module itertools)

    (multiprocessing.pool.multiprocessing.Pool method)

IMAP4

    protocol

IMAP4 (class in imaplib)

IMAP4.abort

IMAP4.error

IOError

ior() (in module operator)

ipow() (in module operator)

ipow() (in module operator)

irepeat() (in module operator)

IRIX

    threads

irshift() (in module operator)

is

    operator, [1]

is not

    operator, [1]

is() (in module operator)

is_alive() (multiprocessing method)

    (threading.Thread method)

is_assigned() (symtable method)

is_blocked()

    (cookielib.DefaultCookie

is_builtin() (in module imp)

is_canonical() (decimal.Context method)

    (decimal.Decimal me

IS_CHARACTER_JUNK

difflib)

is_data() (multifile.MultiFile method)

is_declared_global() (symtable method)

is_empty() (asyncio.fifo method)

is_expired() (cookielib.Cookie method)

is_finite() (decimal.Context method)

    (decimal.Decimal me

is_free() (symtable.Symbol method)

is_frozen() (in module imp)

is_global() (symtable.Symbol method)

is_hop_by_hop() (in module

is_imported() (symtable.
IMAP4.readonly
IMAP4_SSL
  protocol
IMAP4_SSL (class in imaplib)
IMAP4_stream
  protocol
IMAP4_stream (class in imaplib)
imap_unordered()
  (multiprocessing.pool.mp.ProcessPool method)
imaplib (module)
imgfile (module)
imghdr (module)
immedok() (curses.window method)
imutable
  data type
    object, [1], [2]
immutable object
immutable sequence
  object
immutable types
  subclassing
ImmutableSet (class in sets)
imod() (in module operator)
imp
  module
imp (module)
ImplImporter (class in pkgutil)
ImplLoader (class in pkgutil)
import
  statement, [1], [2], [3], [4]
import (2to3 fixer)
import_file() (imputil.DynLoadSuffixImporter method)
import_fresh_module() (in module test/test_support)
IMPORT_FROM (opcode)
import_module() (in module importlib)
method)
is_infinite() (decimal.Context method)
  (decimal.Decimal method)
is_integer() (float method)
is_jython (in module test)
IS_LINE_JUNK() (in module test)
is_linetouched() (curses.window method)
is_local() (symtable.Symbol method)
is_multipart() (email.message.Message method)
is_namespace() (symtable.Symbol method)
is_nan() (decimal.Context method)
  (decimal.Decimal method)
is_nested() (symtable.Symbol method)
is_normal() (decimal.Context method)
  (decimal.Decimal method)
is_not() (in module operator)
is_not_allowed()
  (cookielib.DefaultCookie method)
is_optimized() (symtable.Symbol method)
is_package() (zipimport.zipimporter method)
is_parameter() (symtable.Symbol method)
is_python_build() (in module sysconfig)
is_qnan() (decimal.Context method)
  (decimal.Decimal method)
is_referenced() (symtable.Symbol method)
is_resource_enabled() (in module test.test_support)
is_scriptable() (in module gensuitemodule)
is_set() (threading.Event method)
(in module test.test_support)
IMPORT_NAME (opcode)
IMPORT_STAR (opcode)
import_top() (imputil.Importer method)

importer
Importer (class in imputil)
ImportError
    exception, [1], [2]

importing
importlib (module)
ImportManager (class in imputil)
imports (2to3 fixer)
imports2 (2to3 fixer)
ImportWarning
ImproperConnectionState
imputil (module)
imul() (in module operator)
in
    keyword
    operator, [1], [2]

in_dll() (ctypes._CData method)
in_table_a1() (in module stringprep)
in_table_b1() (in module stringprep)
in_table_c11() (in module stringprep)
in_table_c11_c12() (in module stringprep)
in_table_c12() (in module stringprep)
in_table_c21() (in module stringprep)
in_table_c21_c22() (in module stringprep)
in_table_c22() (in module stringprep)
in_table_c3() (in module stringprep)
in_table_c4() (in module stringprep)
in_table_c5() (in module stringprep)
in_table_c6() (in module stringprep)
in_table_c7() (in module stringprep)
in_table_c8() (in module stringprep)
in_table_c9() (in module stringprep)
in_table_d1() (in module stringprep)
in_table_d2() (in module stringprep)
is_signed() (decimal.Context method)
    (decimal.Decimal method)
is_snan() (decimal.Context method)
    (decimal.Decimal method)
is_subnormal() (decimal method)
    (decimal.Decimal method)
is_tarfile() (in module tar)
is_term_resized() (in module curses)
is_tracked() (in module curses)
is_unverifiable() (urllib2.Request method)
is_wintouched() (curses method)
is_zero() (decimal.Context method)
    (decimal.Decimal method)
is_zipfile() (in module zipfile)
isabs() (in module os.path)
isabstract() (in module inspect)
isAlive() (threading.Thread method)
isalnum() (str method)
isalpha() (str method)
isascii() (str method)
isatty() (file method)
    (in module os)
    (io.IOBase method)
isblank() (in module curses)
isblk() (tarfile.TarInfo method)
isbuiltin() (in module inspect)
isCallable() (in module operator)
ischr() (tarfile.TarInfo method)
isclass() (in module inspect)
iscode() (in module inspect)
inc() (EasyDialogs.ProgressBar method)
inches() (curses.window method)

inclusive

or

Incomplete
IncompleteRead

incr_item(), [1]

increment_lineno() (in module ast)
IncrementalDecoder (class in codecs)
IncrementalEncoder (class in codecs)
IncrementalNewlineDecoder (class in io)
IncrementalParser (class in xml.sax.xmlreader)

indent (doctest.Example attribute)
INDENT (in module token)

indentation

IndentationError

Independent JPEG Group
index (in module cd)
index operation

index() (array.array method)

(is in module operator)
(is in module string)
(list method)
(str method)

(ttk.Notebook method)
(ttk.Treeview method)

IndexError

indexOf() (in module operator)

IndexSizeErr

indices() (slice method)
inet_aton() (in module socket)
inet_ntoa() (in module socket)
inet_ntop() (in module socket)

Inexact (class in decimal)
infile (shlex.shlex attribute)

iscomment() (rfc822.Message method)
isctrl() (in module curses)
isDaemon() (threading.Thread method)
isdatadescriptor() (in module inspect)
isdecimal() (unicode method)
isdev() (tarfile.TarInfo method)
isdigit() (in module curses)

(is in str method)

isdir() (in module os.path)

(tarfile.TarInfo method)
isdisjoint() (set method)

isdown() (in module turtle)
iselement() (in module xml.etree.ElementTree)
isenabled() (in module gc)
isEnabledFor() (logging.Logger method)
isendwin() (in module curses)
ISEOF() (in module tokenize)
isexpr() (in module parser)

(parser.ST method)
isfifo() (tarfile.TarInfo method)
isfile() (in module os.path)

(tarfile.TarInfo method)
isfirstline() (in module file)
isframe() (in module inspect)
isfunction() (in module inspect)
isgenerator() (in module inspect)
isgeneratorfunction() (in module inspect)
isgetsetdescriptor() (in module inspect)
isgraph() (in module curses)
isheader() (rfc822.Message)
isinf() (in modulecmath)

(is in module math)
isinstance (2to3 fixer)
isinstance() (built-in function)
iskeyword() (in module keyword)
islast() (rfc822.Message)
isleap() (in modulecalendar)
Infinity, [1]
info() (gettext.NullTranslations method)
    (in module logging)
    (logging.Logger method)
infoList() (zipfile.ZipFile method)
InfoScrap() (in module Carbon.Scrap)
inheritance
initial() (in module fm)
    (in module mimetypes)
    init_bUILTIN() (in module imp)
    init_color() (in module curses)
    init_database() (in module msilib)
    init_frozen() (in module imp)
    init_pair() (in module curses)
    inited (in module mimetypes)
    initgroups() (in module os)
    initial_indent (textwrap.TextWrapper attribute)
    initialize_options() (distutils.cmd.Command method)
initscr() (in module curses)
INPLACE_ADD (opcode)
INPLACE_AND (opcode)
INPLACE_DIVIDE (opcode)
INPLACE_FLOOR_DIVIDE (opcode)
INPLACE_LSHIFT (opcode)
INPLACE_MODULO (opcode)
INPLACE_MULTIPLY (opcode)
INPLACE_OR (opcode)
INPLACE_POWER (opcode)
INPLACE_RSHIFT (opcode)
INPLACE_SUBTRACT (opcode)
INPLACE_TRUE_DIVIDE (opcode)
INPLACE_XOR (opcode)
input
    built-in function, [1]
    raw
input (2to3 fixer)
input() (built-in function)
    (in module fileinput)
input_charset (email.charset.Charset attribute)
input_charset (email.charset.Charset attribute)
InputOnly (class in Tix)
InputSource (class in xml.sax.xmlreader)
InputType (in module cStringIO)
inquiry (C type)
insch() (curses.window method)
insdelln() (curses.window method)
insert() (array.array method)
    (list method)
    (ttk.Notebook method)
    (ttk.Treeview method)
    (xml.etree.ElementTree.Element method)
insert_text() (in module readline)
insertBefore() (xml.dom.Node method)
InsertionLoc (class in aetypes)
insertln() (curses.window method)
insstr() (curses.window method)
insert() (in module bisect)
insert_left() (in module bisect)
insert_right() (in module bisect)
inspect (module)
instr() (curses.window method)
install() (gettext.NullTranslations method)
    (imputil.ImportManager method)
    (in module gettext)
install_opener() (in module urllib2)
installAutoGIL() (MiniAEFrame.AEServer method)
installAutoGIL() (in module autoGIL)
installHandler() (in module unittest)

instance
    call, [1]
class

isSameNode() (xml.dom
isSequenceType() (in module
isspace() (in module curses
issubclass() (built-in function)
issubset() (set method)
issuite() (in module parser)
    (parser.ST method)
issuperset() (set method)
issym() (tarfile.TarInfo
ISTERMINAL() (in module
istitle() (str method)
istraceback() (in module
issub() (in module operator)
isupper() (in module curses
    (str method)
isvisible() (in module turtle
isxdigit() (in module curses
item
    sequence
    string
item selection
item() (ttk.Treeview method)
    (xml.dom.NamedNodeMap
    (xml.dom.NodeList
    itemgetter() (in module c
    items() (ConfigParser.ConfigParser
        (ConfigParser.RawConfigParser
            (ConfigParser.RawConfigParser
                (ConfigParser.RawConfigParser
                    (ConfigParser.RawConfigParser
                        (email.message.Message
                        (mailbox.Mailbox
                        (xml.etree.ElementTree.Element
                        sequence
                        string
                        item selection
                        item() (ttk.Treeview method)
                        (xml.dom.NamedNodeMap
                        (xml.dom.NodeList
                        itemgetter() (in module config
                        items() (ConfigParser.ConfigParser
                            (ConfigParser.RawConfigParser
                                (ConfigParser.RawConfigParser
                                    (email.message.Message
                                    (mailbox.Mailbox
                                    (xml.etree.ElementTree
                        sequence
                        string
                        item selection
                        item() (ttk.Treeview method)
                        (xml.dom.NamedNodeMap
                        (xml.dom.NodeList
                        itemgetter() (in module config
                        items() (ConfigParser.ConfigParser
                            (ConfigParser.RawConfigParser
                                (ConfigParser.RawConfigParser
                                    (email.message.Message
                                    (mailbox.Mailbox
                                    (xml.etree.ElementTree

object, [1], [2], [3], [4]
instance() (in module new)
instancemethod() (in module new)
InstanceType (in module types)
instate() (ttk.Widget method)
instr() (curses.window method)
istream (shlex.shlex attribute)

int

   built-in function, [1], [2]
int (built-in class)
   (uuid.UUID attribute)
Int2AP() (in module imaplib)
integer
   division
   division, long
   literals
   literals, long
   object, [1], [2]
   representation
   types, operations on

integer division

integer literal
Integral (class in numbers)
Integrated Development Environment
Intel/DVI ADPCM
interact() (code.InteractiveConsole method)
   (in module code)
   (telnetlib.Telnet method)

interact()

interactive

interactive mode
InteractiveConsole (class in code)
InteractiveInterpreter (class in code)
intern (2to3 fixer)
intern() (built-in function)
internal type
internal_attr (zipfile.ZipInfo attribute)
Internaldate2tuple() (in module imaplib)

itemsize (array.array attr
   (memoryview attribute)
ItemsView (class in colle
iter() (built-in function)
   (xml.etree.ElementTr
   (xml.etree.ElementTr
   method)
   method)
iter_child_nodes() (in module
iter_fields() (in module a
iter_importers() (in module
iter_modules() (in modul

iterable

Iterable (class in colle
IterableUserDict (class in

iterator

Iterator (class in collect
iterator protocol
iterdecode() (in module c
iterdump (sqlite3.Connection
iterencode() (in module c
   (json.JSONEncoder
   method)
   (xml.etree.ElementTr
   method)
iteritems() (dict method)
   (mailbox.Mailbox me
iterkeyrefs() 
   (weakref.WeakKeyDictio
iterkeys() (dict method)
   (mailbox.Mailbox me
itermonthdates() (calendar
itermonthdays() (calendar

memoryview attribute)

method)

method)

method)

method)

method)

method)
itermonthdays2() (calendar method)
iterparse() (in module xml.etree.ElementTree)
itertext()
(xml.etree.ElementTree method)
itertools (2to3 fixer)
(module)
itertools_imports (2to3 fixer)
itervaluerefs()
(weakref.WeakValueDictionary method)
itervalues()
(dict method)
(mailbox.Mailbox method)
itertools.Mailbox method)
ITIMER_PROF (in module signal)
ITIMER_REAL (in module signal)
ITIMER_VIRTUAL (in module signal)
TimerError
itruediv() (in module operator)
ixor() (in module operator)
izip() (in module itertools)
izip_longest() (in module itertools)
Index – J

Jansen, Jack
Java
  language
java_ver() (in module platform)
JFIF, [1]
join() (in module os.path)
  (Queue.Queue method)
  (in module string)
  (multiprocessing.JoinableQueue method)
  (multiprocessing.Process method)
  (multiprocessing.pool.multiprocessing.Pool method)
  (str method)
  (threading.Thread method)
join_thread() (multiprocessing.Queue method)
JoinableQueue (class in multiprocessing)
joinfields() (in module string)
jpeg (module)
js_output() (Cookie.BaseCookie method)
  (Cookie.Morsel method)

json
  module
  json (module)
  JSONDecoder (class in json)
  JSONEncoder (class in json)
  JUMP_ABSOLUTE (opcode)
  JUMP_FORWARD (opcode)
  JUMP_IF_FALSE_OR_POP (opcode)
  JUMP_IF_TRUE_OR_POP (opcode)
jumpahead() (in module random)
Index – K

kbhit() (in module msvcrt)
KDEDIR
kevent() (in module select)
key
  (Cookie.Morsel attribute)
key function
key/datum pair
KEY_ALL_ACCESS (in module _winreg)
KEY_CREATE_LINK (in module _winreg)
KEY_CREATE_SUB_KEY (in module _winreg)
KEY_ENUMERATE_SUB_KEYS (in module _winreg)
KEY_EXECUTE (in module _winreg)
KEY_NOTIFY (in module _winreg)
KEY_QUERY_VALUE (in module _winreg)
KEY_READ (in module _winreg)
KEY_SET_VALUE (in module _winreg)
KEY_WOW64_32KEY (in module _winreg)
KEY_WOW64_64KEY (in module _winreg)
KEY_WRITE (in module _winreg)
KeyboardInterrupt
  (built-in exception), [1]
KeyError
keyname() (in module curses)
keypad() (curses.window method)
keyrefs()
  (weakref.WeakKeyDictionary method)
keys() (bsddb.bsddbobject method)
  (dict method)
  (email.message.Message method)
  (mailbox.Mailbox method)
  (sqlite3.Row method)
  (xml.etree.ElementTree.Element method)
keysubst() (in module aetools)
KeysView (class in collections)
keyword
  elif
  else, [1], [2], [3], [4]
  except
  finally, [1], [2], [3], [4]
  from
  in
  yield
Keyword (class in aetypes)
keyword (module)
keyword argument
keywords (functools.partial attribute)
kill() (in module os)
  (subprocess.Popen method)
killchar() (in module curses)
kilpg() (in module os)
knee
module, [1]
knownfiles (in module mimetypes)
kqueue() (in module select)
kwlist (in module keyword)
L (in module re)
label() (EasyDialogs.ProgressBar method)
LabelEntry (class in Tix)
LabelFrame (class in Tix)
**lambda**
    expression, [1]
LambdaType (in module types)
LANG, [1], [2], [3], [4]
LANGUAGE, [1]
language
    C, [1], [2], [3], [4]
    Java
    Pascal
large files
LargeZipFile
last (multifile.MultiFile attribute)
last() (bsddb bsddbobject method)
    (dbhash.dbhash method)
    (nntplib.NNTP method)
last_accepted
    (multiprocessing.connection.Listener attribute)
last_traceback (in module sys), [1]
last_type (in module sys)
last_value (in module sys)
lastChild (xml.dom.Node attribute)
lastcmd (cmd.Cmd attribute)
lastgroup (re.MatchObject attribute)
lastindex (re.MatchObject attribute)
lastpart() (MimeWriter.MimeWriter method)
lastrowid (sqlite3.Cursor attribute)
launch() (in module findertools)
launchurl() (ic.IC method)
    (in module ic)
layout() (ttk.Style method)
literals
    complex number
    floating point
    hexadecimal
    integer
    long integer
    numeric
    octal
LittleEndianStructure (class in ctypes)
ljust() (in module string)
    (str method)
LK_LOCK (in module msvcrt)
    LK_NBLCK (in module msvcrt)
LK_NBRLOCK (in module msvcrt)
LK_RLOCK (in module msvcrt)
LK_UNLCK (in module msvcrt)
LMTP (class in smtplib)
ln() (decimal.Context method)
    (decimal.Decimal method)
LNAME
lngettext() (gettext.GNUTranslations method)
    (gettext.NullTranslations method)
    (in module gettext)
load() (Cookie.BaseCookie method)
    (cookielib.FileCookieJar method)
    (in module hotshot.stats)
    (in module json)
    (in module marshal)
    (in module pickle)
    (pickle.Unpickler method)
LOAD_ATTR (opcode)
load_cert_chain() (ssl.SSLContext method)
LOAD_CLOSURE (opcode)
LBRACE (in module token)
LBYL
LC_ALL, [1]  
   (in module locale)
LC_COLLATE (in module locale)
LC_CTYPE (in module locale)
LC_MESSAGES, [1]  
   (in module locale)
LC_MONETARY (in module locale)
LC_NUMERIC (in module locale)
LC_TIME (in module locale)
lchflags() (in module os)
lchmod() (in module os)
lchown() (in module os)
LDCXXSHARED
ldexp() (in module math)
LDFLAGS
ldgettext() (in module gettext)
ldngettext() (in module gettext)
le() (in module operator)
leading whitespace
leapdays() (in module calendar)
leaveok() (curses.window method)
left (filecmp.dircmp attribute)
left() (in module turtle)
left_list (filecmp.dircmp attribute)
left_only (filecmp.dircmp attribute)
LEFTSHIFT (in module token)
LEFTSHIFTEQUAL (in module token)
len  
   built-in function, [1], [2], [3], [4], [5], [6],
   [7], [8], [9], [10], [11]
len() (built-in function)
length (xml.dom.NamedNodeMap attribute)
   (xml.dom.NodeList attribute)
LESS (in module token)
LESSEQUAL (in module token)
letters (in module string)
(doctest.Example attribute)
(pyclbr.Class attribute)
(pyclbr.Function attribute)
(shlex.shlex attribute)
(xml.parsers.expat.ExpatError attribute)
lineno() (in module fileinput)
LINES, [1], [2]
linesep (in module os)
lineterminator (csv.Dialect attribute)
link() (distutils.ccompiler.CCompiler method)
   (in module os)
link_executable()
   (distutils.ccompiler.CCompiler method)
link_shared_lib()
   (distutils.ccompiler.CCompiler method)
link_shared_object()
   (distutils.ccompiler.CCompiler method)
linkmodel (in module MacOS)
linkname (tarfile.TarInfo attribute)
l脐inux_distribution() (in module platform)
list
   assignment, target
   comprehensions, [1]
deletion target
display
empty
expression, [1], [2]
object, [1], [2], [3], [4], [5], [6], [7], [8]
target, [1]
type, operations on
list (built-in class)
list comprehension
list() (imaplib.IMAP4 method)
   (multiprocessing.managers.SyncManager method)
   (ntplib.NNTP method)
   (in module cmath)
   (in module math)
log1p() (in module math)
log_date_time_string()
   (BaseHTTPServer.BaseHTTPRequestHandler method)
log_error()
   (BaseHTTPServer.BaseHTTPRequestHandler method)
log_exception()
   (wsgiref.handlers.BaseHandler method)
log_message()
   (BaseHTTPServer.BaseHTTPRequestHandler method)
log_request()
   (BaseHTTPServer.BaseHTTPRequestHandler method)
log_to_stderr() (in module multiprocessing)
logb() (decimal.Context method)
Logger (class in logging)
LoggerAdapter (class in logging)
logging
   Errors
logging (module)
logging.config (module)
logging.handlers (module)
Logical (class in aetypes)
logical line
logical_and() (decimal.Context method)
   (decimal.Decimal method)
logical_invert() (decimal.Context method)
   (decimal.Decimal method)
logical_or() (decimal.Context method)
   (decimal.Decimal method)
logical_xor() (decimal.Context method)
login() (ftplib.FTP method)
LIST_APPEND (opcode)
list_dialects() (in module csv)
list_folders() (mailbox.Maildir method)
listallfolders() (mhlib.MH method)
listallsubfolders() (mhlib.MH method)
listdir() (in module dircache)
listfolders() (mhlib.MH method)
listmessages() (mhlib.Folder method)
listMethods() (xmlrpclib.ServerProxy.system method)
Listener (class in multiprocessing.connection)
listfolders() (mhlib.MH method)
listLongMessage (unittest.TestCase attribute)
listLongType (in module types)
lookup() (in module codec)
lookup_error() (in module LookupError)
loop
over mutable sequenc statement, [1], [2], [3]
loop control
target
lower() (in module string)
LPAR (in module token)
lseek() (in module os)
lshift() (in module operator)
LSQB (in module token)
lstat() (in module os)
lstrip() (in module string)
    (str method)
lsub() (imaplib.IMAP4 method)
lt() (in module operator)
    (in module turtle)
Lundh, Fredrik
LWPCookieJar (class in c
Index – M

M (in module re)
mac_ver() (in module platform)
macerrors
    module
macerrors (module)
machine() (in module platform)
MacOS (module)
macostools (module)
macpath (module)
macressource (module)
macros (netrc.netrc attribute)
mailbox
    module
Mailbox (class in mailbox)
mailbox (module)
mailcap (module)
Maildir (class in mailbox)
MaildirMessage (class in mailbox)
MailmanProxy (class in smtpd)
main(), [1]
    (in module py_compile)
    (in module unittest)
mainloop() (FrameWork.Application method)
    (in module turtle)
major() (in module os)
make_archive() (in module distutils.archive_util)
    (in module shutil)
MAKE_CLOSURE (opcode)
make_cookies() (cookielib.CookieJar method)
make_form() (in module fl)
MAKE_FUNCTION (opcode)
make_header() (in module email.header)
make_msgid() (in module email.utils)
make_parser() (in module xml.sax)
methodattrs (2to3 fixer)
methodcaller() (in module methodHelp)
    (xmlrpclib.ServerProxy.s method)
methods
    string
methods (pyclbr.Class a
methodSignature()
    (xmlrpclib.ServerProxy.s method)
MethodType (in module
MH (class in mailbox)
    (class in mhlib)
MHMailbox (class in mail
MHMessage (class in m
microsecond (datetime.c
    (datetime.time attribut
MIME
    base64 encoding
    content type
headers, [1]
    quoted-printable enc
mime_decode_header()
    mimify)
mime_encode_header()
    mimify)
MIMEApplication (class
e-mail.mime.application)
MIMEAudio (class in
e-mail.mime.audio)
MIMEBase (class in
e-mail.mime.base)
make_server() (in module wsgiref.simple_server)
make_tarball() (in module distutils.archive_util)
make_zipfile() (in module distutils.archive_util)
makedev() (in module os)
makedirs() (in module os)
makeelement()
(xml.etree.ElementTree.Element method)
makefile() (socket method)
(socket.socket method)
makefolder() (mhlib.MH method)
makeLogRecord() (in module logging)
makePickle()
(logging.handlers.SocketHandler method)
makeRecord() (logging.Logger method)
makeSocket()
(logging.handlers.DatagramHandler method)
(socket.socket method)
makedev() (in module os)
maketrans() (in module string)
makeusermenus() (FrameWork.Application method)
malloc()
mangling
    name
map (2to3 fixer)
map() (built-in function)
    (in module future_builtins)
    (multiprocessing.pool.multiprocessing.Pool method)
    (ttk.Style method)
map_async()
(multiprocessing.pool.multiprocessing.Pool method)
map_table_b2() (in module stringprep)
map_table_b3() (in module stringprep)
mapcolor() (in module fl)
mapfile() (ic.IC method)
MIMEImage (class in email.mime.image)
MIMEMessage (class in email.mime.message)
MIMEMultipart (class in email.mime.multipart)
MIMENonMultipart (class in email.mime.nonmultipart)
MIMEText (class in email.mime.text)
mimetypes
    module
mimetoos (module)
MimeTypes (class in mimetypes)
MimeWriter (class in MimetWriter)
mimify (module)
mimify() (in module mimify)
min (built-in function)
min (datetime.date attribute)
(min (datetime.datetime attribute)
(min (datetime.timedelta attribute)
(min (decimal.Context method)
(min (decimal.Decimal method)
min_mag() (decimal.Context method)
(min (decimal.Decimal method)
MINEQUAL (in module tokenize)
mapLogRecord() (logging.handlers.HTTPHandler method)
mapping

    mapping (class in collections)
    mapping() (msilib.Control method)
    MappingView (class in collections)
    mapPriority()

(mapLogRecord() (logging.handlers.HTTPHandler method)
(mapLogRecord() (logging.handlers.HTTPHandler method)

Mapping (class in collections)
mapping() (msilib.Control method)
MappingView (class in collections)
mapPriority()

(mapLogRecord() (logging.handlers.HTTPHandler method)
(mapLogRecord() (logging.handlers.HTTPHandler method)

MappingView (class in collections)
mapPriority()

(mapLogRecord() (logging.handlers.HTTPHandler method)
(mapLogRecord() (logging.handlers.HTTPHandler method)

MappingView (class in collections)
mapPriority()

(mapLogRecord() (logging.handlers.HTTPHandler method)
(mapLogRecord() (logging.handlers.HTTPHandler method)

MappingView (class in collections)
mapPriority()

(mapLogRecord() (logging.handlers.HTTPHandler method)
(mapLogRecord() (logging.handlers.HTTPHandler method)

MappingView (class in collections)
mapPriority()

(mapLogRecord() (logging.handlers.HTTPHandler method)
(mapLogRecord() (logging.handlers.HTTPHandler method)

MappingView (class in collections)
mapPriority()

(mapLogRecord() (logging.handlers.HTTPHandler method)
(mapLogRecord() (logging.handlers.HTTPHandler method)

MappingView (class in collections)
mapPriority()

(mapLogRecord() (logging.handlers.HTTPHandler method)
(mapLogRecord() (logging.handlers.HTTPHandler method)

MappingView (class in collections)
mapPriority()

(mapLogRecord() (logging.handlers.HTTPHandler method)
(mapLogRecord() (logging.handlers.HTTPHandler method)

MappingView (class in collections)
mapPriority()

(mapLogRecord() (logging.handlers.HTTPHandler method)
(mapLogRecord() (logging.handlers.HTTPHandler method)

MappingView (class in collections)
mapPriority()

(mapLogRecord() (logging.handlers.HTTPHandler method)
(mapLogRecord() (logging.handlers.HTTPHandler method)

MappingView (class in collections)
mapPriority()

(mapLogRecord() (logging.handlers.HTTPHandler method)
(mapLogRecord() (logging.handlers.HTTPHandler method)

MappingView (class in collections)
mapPriority()
MAX_INTERPOLATION_DEPTH (in module ConfigParser)
max_mag() (decimal.Context method)
    (decimal.Decimal method)
maxarray (repr.Repr attribute)
maxdeque (repr.Repr attribute)
maxdict (repr.Repr attribute)
maxDiff (unittest.TestCase attribute)
maxfrozenset (repr.Repr attribute)
maxint (in module sys)
maxlen (collections.deque attribute)
MAXLEN (in module mimify)
maxlevel (repr.Repr attribute)
maxlist (repr.Repr attribute)
maxlong (repr.Repr attribute)
maxother (repr.Repr attribute)
maxpp() (in module audioop)
maxset (repr.Repr attribute)
maxsize (in module sys)
maxstring (repr.Repr attribute)
maxtuple (repr.Repr attribute)
maxunicode (in module sys)
maxval (EasyDialogs.ProgressBar attribute)
MAXYEAR (in module datetime)
MB_ICONASTERISK (in module winsound)
MB_ICONEXCLAMATION (in module winsound)
MB_ICONHAND (in module winsound)
MB_ICONQUESTION (in module winsound)
MB_OK (in module winsound)
mbox (class in mailbox)
mboxMessage (class in mailbox)
md5 (module)
md5() (in module md5)
MemberDescriptorType (in module types)
membership
    test
memmove() (in module ctypes)
MemoryError
MemoryHandler (class in logging.handlers)
memoryview (built-in class)
memset() (in module ctypes)
Menu() (in module FrameWork)
MenuBar() (in module FrameWork)
Menuitem() (in module FrameWork)
merge() (in module heapq)
Message (class in email.message)
    (in module mailbox)
    (class in mlib)
    (class in mimetools)
    (class in rfc822)
    (in module mimetools)
message digest, MD5, [1]
Message() (in module EasyDialogs)
message_from_file() (in module email)
message_from_string() (in module email)
MessageBeep() (in module winsound)
MessageClass
    (BaseHTTPServer.BaseHTTPRequestHandler attribute)
MessageError
MessageParseError
meta() (in module curses)
meta_path (in module sys)
    metaclass
        (2to3 fixer)
metavar (optparse.Option attribute)
Meter (class in Tix)
    METH_CLASS (built-in variable)
    METH_COEXIST (built-in variable)
    METH_KEYWORDS (built-in variable)
    METH_NOARGS (built-in variable)
    METH_O (built-in variable)
    METH_OLDARGS (built-in variable)
    METH_STATIC (built-in variable)
    METH_VARARGS (built-in variable)
htmlib
icglue
ihooks
imp
importing
json
knee, [1]
macerors
mailbox
math, [1]
mimetools
namespace
object, [1], [2]
os, [1]
pickle, [1], [2], [3], [4]
pty
pwd
pyexpat
re, [1], [2]
readline
reexe
rfc822
rlcompleter
search path, [1], [2], [3], [4]
sgmlib
shelve
signal, [1]
sitecustomize
socket, [1]
stat
statvfs
string, [1], [2]
struct
sunaudiodev
method
  built-in
  call
  object, [1], [2], [3], [4], [5]
user-defined
method resolution order

sys, [1], [2], [3], [4], [5]
thread
types
urllib
 urlparse
usercustomize
uu
module (pyclbr.Class attribute)
(pyclbr.Function attribute)
module() (in module new)
ModuleFinder (class in modulefinder)
modulefinder (module)
modules (in module sys)
  (modulefinder.ModuleFinder attribute)
ModuleType (in module types)
modulo
mono2grey() (in module imageop)
month (datetime.date attribute)
  (datetime.datetime attribute)
month() (in module calendar)
month_abbr (in module calendar)
month_name (in module calendar)
monthcalendar() (in module calendar)
monthdatescalendar() (calendar.Calendar method)
monthdays2calendar() (calendar.Calendar method)
monthdayscalendar() (calendar.Calendar method)
monthrange() (in module calendar)
Morsel (class in Cookie)
most_common() (collections.Counter method)
mouseinterval() (in module curses)
mousemask() (in module curses)
movе() (curses.panel.Pa...
(curses.window method)
(in module findertools)
(in module shutil)
(mmap.mmap method)
(ttk.Treeview method)
move_file()
(distutils.ccompiler.CCompiler method)
(movemessage() (mhlib.Folder method)
MozillaCookieJar (class MRO)
mro() (class method)
msftoframe() (in module msg (httplib.HTTPResponse)
msg() (telnetlib.Telnet method)
msi
msilib (module)
msvcrt (module)
mt_interact() (telnetlib.Telnet method)
mtime (tarfile.TarInfo attribute)
mtime() (robotparser.RobotFileParser method)
mul() (in module audioop)
(in module operator)
MultiCall (class in xmlrpclib)
MultiFile (class in multifile)
multifile (module)
MULTILINE (in module re)
MultipartConversionError
multiplication
multiply() (decimal.Context method)
multiprocessing (module)
multiprocessing.connect
multiprocessing.dummy
multiprocessing.Manager
multiprocessing.sharedctypes
mutable

object, [1], [2]

sequence types

mutable object

mutable sequence

loop over

object

MutableMapping (class in collections)

MutableSequence (class in collections)

MutableSet (class in collections)

MutableString (class in collections)

mutex (class in mutex)

(module)

mvderwin() (curses.window)

mvwin() (curses.window)

myrights() (imaplib.IMAP
Index – N

N_TOKENS (in module token)
name, [1], [2]
  binding, [1], [2], [3], [4], [5]
  binding, global
  class
  function
  mangling
  rebinding
  unbinding
name (cookielib.Cookie attribute)
  (doctest.DocTest attribute)
  (file attribute)
  (in module os)
NAME (in module token)
name (io.FileIO attribute)
  (multiprocessing.Process attribute)
  (ossaudiodev.oss_audio_device attribute)
  (pyclbr.Class attribute)
  (pyclbr.Function attribute)
  (tarfile.TarInfo attribute)
  (threading.Thread attribute)
  (xml.dom.Attr attribute)
  (xml.dom.DocumentType attribute)
name() (in module unicodedata)
name2codepoint (in module htmlentitydefs)
**named tuple**
NamedTemporaryFile() (in module tempfile)
namedtuple() (in module collections)
NameError
  exception
  NNTPTemporaryError
no_proxy, [1]
nocbreak() (in module curses)
NoDataAllowedErr
Node (class in compiler.ast)
node() (in module platform)
nodelay() (curses.window method)
nodeName (xml.dom.Node attribute)
NodeTransformer (class in ast)
nodetype (xml.dom.Node attribute)
nodeValue (xml.dom.Node attribute)
NodeVisitor (class in ast)
NODISC (in module cd)
noecho() (in module curses)
NOEXPR (in module locale)
nofill (httplib.HTTPParser attribute)
nok_builtin_names
  (rexec.RExec attribute)
noload() (pickle.Unpickler method)
NoModificationAllowedErr
nonblock()
  (ossaudiodev.oss_audio_device method)
**None**
  object, [1], [2]
None (Built-in object)
  (built-in variable)
NoneType (in module types)
nonl() (in module curses)
NameError (built-in exception)
namelist() (zipfile.ZipFile method)
nameprep() (in module encodings.idna)

names
  private
namespace, [1]
  global
module
Namespace (class in argparse)
  (class in multiprocessing.managers)
namespace() (imaplib.IMAP4 method)
Namespace()
  (multiprocessing.managers.SyncManager method)
NAMESPACE_DNS (in module uuid)
NAMESPACE_OID (in module uuid)
NAMESPACE_URL (in module uuid)
NAMESPACE_X500 (in module uuid)
NamespaceErr
namespaceURI (xml.dom.Node attribute)

NaN, [1]
NannyNag
napms() (in module curses)
nargs (optparse.Option attribute)
Nav (module)
Navigation Services
ndiff() (in module difflib)
ndim (memoryview attribute)
ne (2to3 fixer)
ne() (in module operator)
neg() (in module operator)
negation

nested scope
nested() (in module contextlib)
netrc (class in netrc)
  (module)
NetrcParseError
netscape (cookielib.CookiePolicy
nonzero (2to3 fixer)
noop() (imaplib.IMAP4 method)
  (poplib.POP3 method)
NoOptionError
NOP (opcode)
oqiflush() (in module curses)
oraw() (in module curses)
normalize() (decimal.Decimal method)
  (decimal.Decimal method)
  (in module locale)
  (in module unicodedata)
  (xml.dom.Node method)
NORMALIZE_WHITESPACE
  (in module doctest)
normvariate() (in module random)
normcase() (in module os.pat
normpath() (in module os.pat
NoSectionError
NoSuchMailboxError
not
  operator, [1]
not in
  operator, [1], [2]
not_() (in module operator)
NotANumber
notation
notationDecl()
  (xml.sax.handler.DTDHandle method)
NotationDeclHandler()
  (xml.parsers.expat.xmlparser method)
notations
  (xml.dom.DocumentType attribute)
NotConnected
attribute)
Network News Transfer Protocol
new (module)
new() (in module hmac)
    (in module md5)
    (in module sha)

**new-style class**
new_alignment() (formatter.writer method)
new_compiler() (in module distutils.ccompiler)
new_font() (formatter.writer method)
new_margin() (formatter.writer method)
new_module() (in module imp)
new_panel() (in module curses.panel)
new_spacing() (formatter.writer method)
new_styles() (formatter.writer method)
newconfig() (in module al)
newer() (in module distutils.dep_util)
newer_group() (in module distutils.dep_util)
newer_pairwise() (in module distutils.dep_util)
newgroups() (nntplib.NNTP method)

**newline**
    suppression
NEWLINE (in module token)
NEWLINE token, [1]
newlines (file attribute)
    (io.TextIOBase attribute)
newnews() (nntplib.NNTP method)
newpad() (in module curses)
newwin() (in module curses)
next (2to3 fixer)
next() (bsddb.bsddbobject method)
    (built-in function)
    (csv.csvreader method)
    (dbhash.dbhash method)

NoteBook (class in Tix)
Notebook (class in ttk)
NotEmptyError
NOTEQUAL (in module token)
NotFoundErr
notify() (threading.Condition method)
notify_all() (threading.Condition method)
notifyAll() (threading.Condition method)
notimeout() (curses.window method)

NotImplemented
    object
NotImplemented (built-in variable)
NotImplementedError
NotImplementedType (in module types)
NotStandaloneHandler()
    (xml.parsers.expat.xmlparser method)
NotSupportedErr
noutrefresh() (curses.window method)
now() (datetime.datetime class method)

NProperty (class in aetypes)
NSIG (in module signal)
nsmallest() (in module heapq)
NT_OFFSET (in module token)
NTEventLogHandler (class in logging.handlers)
ntohl() (in module socket)
ntohs() (in module socket)
ntransfercmd() (ftplib.FTP method)
null
NullFormatter (class in formatter)
NullHandler (class in logging)
NullImporter (class in imp)
NullTranslations (class in gettext)
NullWriter (class in formatter)

complex

floating point

Number (class in numbers)
NUMBER (in module token)
number_class()

(decimal.Context method)
(decimal.Decimal method)

numbers (module)

numerator (numbers.Rational attribute)

numeric

calculations

literals

object, [1], [2], [3], [4]

types, operations on

numeric literal

numeric() (in module unicodedata)

Numerical Python

numliterals (2to3 fixer)
nurbscurve() (in module gl)
nurbssurface() (in module gl)
nvarray() (in module gl)

NNTP

protocol

NNTP (class in nntplib)

NNTPDataError

NNTPError

nntplib (module)
Index – O

O_APPEND (in module os)
O_ASYNC (in module os)
O_BINARY (in module os)
O_CREAT (in module os)
O_DIRECT (in module os)
O_DIRECTORY (in module os)
O_DSYNC (in module os)
O_EXCL (in module os)
O_EXLOCK (in module os)
O_NDELAY (in module os)
O_NOATIME (in module os)
O_NOCTTY (in module os)
O_NOFOLLOW (in module os)
O_NOINHERIT (in module os)
O_NONBLOCK (in module os)
O_RANDOM (in module os)
O_RDONLY (in module os)
O_RDWR (in module os)
O_RSYNC (in module os)
O_SEQUENTIAL (in module os)
O_SHLOCK (in module os)
O_SHORT_LIVED (in module os)
O_SYNC (in module os)
O_TEMPORARY (in module os)
O_TCPNOP (in module os)
O_TRUNC (in module os)
O_WRONLY (in module os)

object, [1]
  Boolean, [1]
  CObject
  Capsule
  Ellipsis
  None, [1], [2]
  NotImplemented
  Unicode

open_osfhandle() (in module msvcr)
open_unknown()
(urllib.URLopener method)
OpenDatabase() (in module msilib)
opendir() (in module dircache)
OpenerDirector (class in urllib2)
openfolder() (mhib.MH method)
openfp() (in module sunau)
  (in module wave)
OpenGL
OpenKey() (in module _winreg)
OpenKeyEx() (in module _winreg)
openlog() (in module syslog)
openmessage() (mhib.Message method)
openmixer() (in module ossaudiodev)
openport() (in module al)
openpty() (in module os)
  (in module pty)
openrf() (in module MacOS)

OpenSSL
  (use in module hashlib)
  (use in module ssl)
OPENSSL_VERSION (in module ssl)
OPENSSL_VERSION_INFO (module ssl)
OPENSSL_VERSION_NUMB (in module ssl)
OpenView() (msilib.Database method)
buffer, [1]
built-in function, [1]
built-in method, [1]
bytearray, [1]
callable, [1]
class, [1], [2], [3], [4]
class instance, [1], [2], [3]
code, [1], [2], [3]
complex
complex number, [1]
deallocation
dictionary, [1], [2], [3], [4], [5], [6], [7]
file, [1], [2], [3], [4]
finalization
floating point, [1], [2]
frame
frozenset, [1]
function, [1], [2], [3], [4], [5]
generator, [1], [2]
immutable, [1], [2]
immutable sequence
instance, [1], [2], [3], [4]
integer, [1], [2]
list, [1], [2], [3], [4], [5], [6], [7], [8]
long integer, [1], [2]
mapping, [1], [2], [3], [4], [5]
method, [1], [2], [3], [4], [5]
module, [1], [2]
mutable, [1], [2]
mutable sequence
numeric, [1], [2], [3], [4]
plain integer
recursive
sequence, [1], [2], [3], [4], [5], [6], [7],

operation
  Boolean
  binary arithmetic
  binary bitwise
  concatenation
  extended slice
  null
  repetition
  shifting
  slice
  subscript
  unary arithmetic
  unary bitwise

operations
  Boolean, [1]
  bitwise
  masking
  shifting

operations on
  dictionary type
  integer types
  list type
  mapping types
  numeric types
  sequence types, [1]

operator
  !=
  %
  &
  *
  **
  +
  -
  /
  //
set, [1], [2], [3]
set type
slice
socket
string, [1], [2], [3], [4]
traceback, [1], [2], [3], [4]
tuple, [1], [2], [3], [4], [5]
type, [1], [2]
unicode
user-defined function, [1], [2]
user-defined method
xrange, [1]
object (built-in class)
  (exceptions.UnicodeError attribute)
object_filenames()
  (distutils.ccompiler.CCompiler method)
objects
  comparing
  flattening
  marshalling
  persistent
  pickling
  serializing
ObjectSpecifier (class in aetypes)
obufcount()
  (ossaudiodev.oss_audio_device method)
obuffree()
  (ossaudiodev.oss_audio_device method)
oct
  built-in function
oct() (built-in function)
  (in module future_builtins)
octal
<
<<
<=
==
>
>=
>>
^ and, [1], [2]
comparison
in, [1], [2]
is, [1]
is not, [1]
not, [1]
not in, [1], [2]
or, [1], [2]
overloading
precedence
ternary
operator (module)
operators
opmap (in module dis)
opname (in module dis)
optimize() (in module pickletoc)
OptionGroup (class in optpars)
OptionMenu (class in Tix)
OptionParser (class in optpars)
options (doctest.Example attribute)
  (ssl.SSLContext attribute)
options()
  (ConfigParser.RawConfigParser method)
optionxform()
  (ConfigParser.RawConfigParser method)
literals
octal literal
octdigits (in module string)
offset (xml.parsers.expat.ExpatError attribute)
OK (in module curses)
ok_builtin_modules (rexe.RExec attribute)
ok_file_types (rexe.RExec attribute)
ok_path (rexe.RExec attribute)
ok_posix_names (rexe.RExec attribute)
ok_sys_names (rexe.RExec attribute)
OleDLL (class in ctypes)
onclick() (in module turtle), [1]
ondrag() (in module turtle)
onecmd() (cmd.Cmd method)
onkey() (in module turtle)
onrelease() (in module turtle)
onscreenclick() (in module turtle)
tonimer() (in module turtle)
OP (in module token)
OP_ALL (in module ssl)
OP_CIPHER_SERVER_PREFERENCE (in module ssl)
OP_NO_COMPRESSION (in module ssl)
OP_NO_SSLv2 (in module ssl)
OP_NO_SSLv3 (in module ssl)
OP_NO_TLSv1 (in module ssl)
OP_NO_TLSv1_1 (in module ssl)
OP_NO_TLSv1_2 (in module ssl)
OP_SINGLE_DH_USE (in module ssl)
OP_SINGLE_ECDH_USE (in module ssl)
open
    built-in function, [1]
Open Scripting Architecture
optparse (module)
or
    bitwise
    exclusive
    inclusive
    operator, [1], [2]
or_() (in module operator)
ord
    built-in function, [1], [2]
ord() (built-in function)
order
evaluation
ordered_attributes
(xml.parsers.expat.xmlparser attribute)
OrderedDict (class in collections)
Ordinal (class in aetypes)
origin_server
(wsgiref.handlers.BaseHandler attribute)
os
    module, [1]
os (module)
os.path (module)
os.environ
(wsgiref.handlers.BaseHandler attribute)
OSError
ossaudiodev (module)
OSSAudioError
output, [1]
    standard, [1]
output
(subprocess.CalledProcessError attribute)
output() (Cookie.BaseCookie method)
open() (built-in function)
  (FrameWork.DialogWindow method)
  (FrameWork.Window method)
  (distutils.text_file.TextFile method)
  (imaplib.IMAP4 method)
  (in module aifc)
  (in module anydbm)
  (in module cd)
  (in module codecs)
  (in module dbhash)
  (in module dbm)
  (in module dl)
  (in module dumbdbm)
  (in module gdbm)
  (in module gzip)
  (in module io)
  (in module os)
  (in module ossaudiodev)
  (in module posixfile)
  (in module shelve)
  (in module sunau)
  (in module sunaudiodev)
  (in module tarfile)
  (in module wave)
  (in module webbrowser)
  (pipes.Template method)
  (tarfile.TarFile class method)
  (telnetlib.Telnet method)
  (urllib.URLopener method)
  (urllib2.OpenerDirector method)
  (webbrowser.controller method)
  (zipfile.ZipFile method)
open_new() (in module webbrowser)
  (webbrowser.controller method)

(output_charset)
  (email.charset.Charset attribute)
output_charset()
  (gettext.NullTranslations method)
output_codec
  (email.charset.Charset attribute)
output_difference()
  (doctest.OutputChecker method)
OutputChecker (class in doctest)
OutputString() (Cookie.Morsel method)
OutputType (in module cStringIO)
Overflow (class in decimal)
OverflowError
  (built-in exception), [1],
  [3], [4], [5]
overlay() (curses.window method)

overloading
  operator

Overmars, Mark
overwrite() (curses.window method)
open_new_tab() (in module webbrowser)
   (webbrowser.controller method)
Index – P

P_DETACH (in module os)  PyExc_Exception
P_NOWAIT (in module os)  PyExc_FloatingPointError
P_NOWAITO (in module os)  PyExc_ImportError
P_OVERLAY (in module os)  PyExc_IndexError
P_WAIT (in module os)  PyExc_IOError
pack() (in module aepack)  PyExc_KeyboardInterrupt
   (in module struct)
   (mailbox.MH method)
   (struct.Struct method)
pack_array() (xdrlib.Packer method)
pack_bytes() (xdrlib.Packer method)
pack_double() (xdrlib.Packer method)
pack_farray() (xdrlib.Packer method)
pack_float() (xdrlib.Packer method)
pack_fopaque() (xdrlib.Packer method)
pack_fstring() (xdrlib.Packer method)
pack_into() (in module struct)
   (struct.Struct method)
pack_list() (xdrlib.Packer method)
pack_opaque() (xdrlib.Packer method)
pack_string() (xdrlib.Packer method)
package, [1], [2]

package variable
   __all__

Packer (class in xdrlib)
packevent() (in module aetools)
packing
   binary data
packing (widgets)
PAGER, [1]
pair_content() (in module curses)
pair_number() (in module curses)
PanedWindow (class in Tix)

parameter
   call semantics

pyexpat
   module
   PyFile_AsFile ()
   PyFile_Check ()
   PyFile_CheckErr
   PyFile_DecUseCount
   PyFile_FromFile
   PyFile_FromString
   PyFile_GetLine
   PyFile_IncUseCount
   PyFile_Name
   PyFile_SetBufSize
   PyFile_SetEncoding
difference from argument
function definition
value, default
pardir (in module os)
paren (2to3 fixer)
parent (urllib2.BaseHandler attribute)
parent() (ttk.Treeview method)
parenthesized form
parentNode (xml.dom.Node attribute)
paretovariate() (in module random)
parse() (doctest.DocTestParser method)
  (email.parser.Parser method)
  (in module ast)
  (in module cgi)
  (in module compiler)
  (in module xml.dom.minidom)
  (in module xml.dom.pulldom)
  (in module xml.etree.ElementTree)
  (in module xml.sax)
  (robotparser.RobotFileParser method)
  (string.Formatter method)
  (xml.etree.ElementTree.ElementTree method)
  Parse() (xml.parsers.expat.xmlparser method)
parse() (xml.sax.xmlreader.XMLReader method)
parse_and_bind() (in module readline)
parse_args() (argparse.ArgumentParser method)
PARSE_COLNAMES (in module sqlite3)
parse_config_h() (in module sysconfig)
PARSE_DECLTYPES (in module sqlite3)
parse_header() (in module cgi)
parse_known_args() (argparse.ArgumentParser method)
parse_multipart() (in module cgi)
parse_qs() (in module cgi)
  (in module urlparse)
parse_qsl() (in module cgi)
  (in module urlparse)
parseaddr() (in module email.utils)
    (in module rfc822)
parsedate() (in module email.utils)
    (in module rfc822)
parsedate_tz() (in module email.utils)
    (in module rfc822)
parseFile() (in module compiler)
ParseFile() (xml.parsers.expat.xmlparser method)
ParseFlags() (in module imaplib)
parser
Parser (class in email.parser)
parser (module)
ParserCreate() (in module xml.parsers.expat)
ParserError
ParseResult (class in urlparse)
parsesequence() (mhlib.Folder method)
parsestr() (email.parser.Parser method)
parseString() (in module xml.dom.minidom)
    (in module xml.dom.pulldom)
    (in module xml.sax)
parseurl() (ic.IC method)
    (in module ic)
parsing
    Python source code
    URL
ParsingError
partial() (imaplib.IMAP4 method)
    (in module functools)
partition() (str method)
Pascal
    language
pass
    statement
pass_() (poplib.POP3 method)
Paste
PATH, [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12],
[13], [14], [15], [16], [17], [18]
path
configuration file
module search, [1], [2], [3], [4], [5], [6]
operations
path (BaseHTTPServer.BaseHTTPRequestHandler attribute)
(cookielib.Cookie attribute)
in module sys), [1], [2], [3]
Path browser
path_hooks (in module sys)
path_importer_cache (in module sys)
path_return_ok() (cookielib.CookiePolicy method)
pathconf() (in module os)
pathconf_names (in module os)
pathname2url() (in module urllib)
pathsep (in module os)
pattern (re.RegexObject attribute)
pause() (in module signal)
PAUSED (in module cd)
PAX_FORMAT (in module tarfile)
pax_headers (tarfile.TarFile attribute)
(tarfile.TarInfo attribute)
pbkdf2_hmac() (in module hashlib)
pd() (in module turtle)
Pdb (class in pdb), [1]
pdb (module)
peek() (io.BufferedReader method)
PEM_cert_to_DER_cert() (in module ssl)
pen() (in module turtle)
pencolor() (in module turtle)
PendingDeprecationWarning
pendown() (in module turtle)
pensize() (in module turtle)
penup() (in module turtle)
PERCENT (in module token)
PERCENTEQUAL (in module token)
Performance
permutations() (in module itertools)
Persist() (msilib.SummaryInformation method)
persistence
persistent
  objects
persistent_id (pickle protocol)
persistent_load (pickle protocol)
pformat() (in module pprint)
    (pprint.PrettyPrinter method)
phase() (in modulecmath)
Philbrick, Geoff
physical line, [1], [2]
pi (in modulecmath)
  (in module math)
pick() (in modulegl)
pickle
  module, [1], [2], [3], [4]
pickle (module)
pickle() (in modulecopy_reg)
PickleError
Pickler (class in pickle)
pickletools (module)
pickling
  objects
PicklingError
pid (multiprocessing.Process attribute)
  (popen2.Popen3 attribute)
  (subprocess.Popen attribute)
PIL (the Python Imaging Library)
PIPE (in module subprocess)
Pipe() (in module multiprocessing)
pipe() (in moduleos)
PIPE_BUF (select.select attribute)
pipes (module)
PixMapWrapper (module)
PKG_DIRECTORY (in moduleimp)
pkgutil (module)
plain integer
  object
plain integer literal
PLAT
platform (in module sys), [1]  
(module)
platform() (in module platform)
PLAYING (in module cd)
PlaySound() (in module winsound)
plist
  file
plistlib (module)
plock() (in module os)
plus
PLUS (in module token)
plus() (decimal.Context method)
PLUSEQUAL (in module token)
pm() (in module pdb)
num (in module cd)
POINTER() (in module ctypes)
pointer() (in module ctypes)
polar() (in module cmath)
poll() (in module select)
  (multiprocessing.Connection method)
  (popen2.Popen3 method)
  (select.epoll method)
  (select.poll method)
  (subprocess.Popen method)
pop() (array.array method)
  (asynchat.fifo method)
  (collections.deque method)
  (dict method)
  (list method)
  (mailbox.Mailbox method)
  (multifile.MultiFile method)
  (set method)
POP3
  protocol
POP3 (class in poplib)
POP3_SSL (class in poplib)

pop_alignment() (formatter.formatter method)

POP_BLOCK (opcode)

pop_font() (formatter.formatter method)

POP_JUMP_IF_FALSE (opcode)

POP_JUMP_IF_TRUE (opcode)

pop_margin() (formatter.formatter method)

pop_source() (shlex.shlex method)

pop_style() (formatter.formatter method)

POP_TOP (opcode)

Popen (class in subprocess)

popen() (in module os), [1], [2]

   (in module platform)

popen2 (module)

popen2() (in module os)

   (in module popen2)

Popen3 (class in popen2)

popen3() (in module os)

   (in module popen2)

Popen4 (class in popen2)

popen4() (in module os)

   (in module popen2)

popitem() (collections.OrderedDict method)

   (dict method)

   (mailbox.Mailbox method)

popleft() (collections.deque method)

poplib (module)

PopupMenu (class in Tix)

port (cookielib.Cookie attribute)

port_specified (cookielib.Cookie attribute)

PortableUnixMailbox (class in mailbox)

pos (re.MatchObject attribute)

pos() (in module operator)

   (in module turtle)

position() (in module turtle)

position argument

POSIX

   I/O control
file object
threads
posix (module)
  (tarfile.TarFile attribute)
posixfile (module)
POSIXLY_CORRECT
post() (ntplib.NNTP method)
  (ossaudiodev.oss_audio_device method)
post_mortem() (in module pdb)
postcmd() (cmd.Cmd method)
postloop() (cmd.Cmd method)
pow
  built-in function, [1], [2], [3], [4], [5]
pow() (built-in function)
  (in module math)
  (in module operator)
power() (decimal.Context method)
pprint (module)
pprint() (bdb.Breakpoint method)
  (in module pprint)
  (pprint.PrettyPrinter method)
prcal() (in module calendar)
preamble (email.message.Message attribute)
precedence
  operator
precmd() (cmd.Cmd method)
prefix, [1], [2], [3]
PREFIX (in module distutils.sysconfig)
prefix (in module sys)
  (xml.dom.Attr attribute)
  (xml.dom.Node attribute)
  (zipimport.zipimporter attribute)
PREFIXES (in module site)
preloop() (cmd.Cmd method)
preorder() (compiler.visitor.ASTVisitor method)
prepare_input_source() (in module xml.sax.saxutils)
prepend() (pipes.Template method)
preprocess() (distutils.ccompiler.CCompiler method)
PrettyPrinter (class in pprint)
prev() (ttk.Treeview method)
previous() (bsddb.bsddbobject method)
    (dbhash.dbhash method)
previousSibling (xml.dom.Node attribute)
primary
    statement, [1], [2]
print (2to3 fixer)
print() (built-in function)
Print() (in module findertools)
print_callees() (pstats.Stats method)
print_callers() (pstats.Stats method)
print_directory() (in module cgi)
print_environ() (in module cgi)
print_environ_usage() (in module cgi)
print_exc() (in module traceback)
    (timeit.Timer method)
print_exception() (in module traceback)
PRINT_EXPR (opcode)
print_form() (in module cgi)
print_help() (argparse.ArgumentParser method)
PRINT_ITEM (opcode)
PRINT_ITEM_TO (opcode)
print_last() (in module traceback)
PRINT_NEWLINE (opcode)
PRINT_NEWLINE_TO (opcode)
print_stack() (in module traceback)
print_stats() (profile.Profile method)
    (pstats.Stats method)
print_tb() (in module traceback)
print_usage() (argparse.ArgumentParser method)
    (optparse.OptionParser method)
print_version() (optparse.OptionParser method)
printable (in module string)
printdir() (zipfile.ZipFile method)
printf-style formatting
PriorityQueue (class in Queue)

private
    names

prmonth() (calendar.TextCalendar method)
    (in module calendar)

procedure
call

process
    group, [1]
    id
    id of parent
    killing, [1]
    signalling, [1]

Process (class in multiprocessing)

process() (logging.LoggerAdapter method)

process_message() (smtplib.SMTPServer method)

process_request() (SocketServer.BaseServer method)

processes, light-weight

processfile() (in module gensuitemodule)

processfile_fromresource() (in module gensuitemodule)

ProcessingInstruction() (in module xml.etree.ElementTree)

processingInstruction()
    (xml.sax.handler.ContentHandler method)

ProcessingInstructionHandler()
    (xml.parsers.expat.xmlparser method)

processor time

processor() (in module platform)

product() (in module itertools)

Profile (class in hotshot)
    (class in profile)

profile (module)

profile function, [1], [2]

profiler, [1]

profiling, deterministic

program
Progressbar (class in ttk)
ProgressBar() (in module EasyDialogs)
prompt (cmd.Cmd attribute)
prompt_user_passwd() (urllib.FancyURLopener method)
prompts, interpreter
propagate (logging.Logger attribute)
property (built-in class)
property list
property_declaration_handler (in module xml.sax.handler)
property_dom_node (in module xml.sax.handler)
property_lexical_handler (in module xml.sax.handler)
property_xml_string (in module xml.sax.handler)
prot_c() (ftplib.FTP_TLS method)
prot_p() (ftplib.FTP_TLS method)
proto (socket.socket attribute)
protocol
  CGI
  FTP, [1]
  HTTP, [1], [2], [3], [4]
  IMAP4
  IMAP4_SSL
  IMAP4_stream
  NNTP
  POP3
  SMTP
  Telnet
    context management
    iterator
protocol (ssl.SSLContext attribute)
PROTOCOL_SSLv2 (in module ssl)
PROTOCOL_SSLv23 (in module ssl)
PROTOCOL_SSLv3 (in module ssl)
PROTOCOL_TLS (in module ssl)
PROTOCOL_TLSv1 (in module ssl)
PROTOCOL_TLSv1_1 (in module ssl)
PROTOCOL_TLSv1_2 (in module ssl)
protocol_version
(BaseHTTPServer.BaseHTTPRequestHandler attribute)
PROTOCOL_VERSION (imaplib.IMAP4 attribute)
ProtocolError (class in xmlrpclib)
proxy() (in module weakref)
proxyauth() (imaplib.IMAP4 method)
ProxyBasicAuthHandler (class in urllib2)
ProxyDigestAuthHandler (class in urllib2)
ProxyHandler (class in urllib2)
ProxyType (in module weakref)
ProxyTypes (in module weakref)
prstr() (in module fm)
pryear() (calendar.TextCalendar method)
ps1 (in module sys)
ps2 (in module sys)
pstats (module)
pthreads
ptime (in module cd)

pty
    module
pty (module)
pu() (in module turtle)
publicId (xml.dom.DocumentType attribute)
PullDOM (class in xml.dom.pulldom)
punctuation (in module string)
PureProxy (class in smtpd)
purge() (in module re)
Purpose.CLIENT_AUTH (in module ssl)
Purpose.SERVER_AUTH (in module ssl)
push() (asynchat.async_chat method)
    (asynchat.fifo method)
    (code.InteractiveConsole method)
    (multifile.MultiFile method)
push_alignment() (formatter.formatter method)
push_font() (formatter.formatter method)
push_margin() (formatter.formatter method)
push_source() (shlex.shlex method)
push_style() (formatter.formatter method)
push_token() (shlex.shlex method)
push_with_producer() (asynchat.async_chat method)
pushbutton() (msilib.Dialog method)
put()
(multiprocessing.multiprocessing.queues.SimpleQueue method)
(Queue.Queue method)
(multiprocessing.Queue method)
put_nowait() (multiprocessing.Queue method)
(Queue.Queue method)
putch() (in module msvcrt)
putenv() (in module os)
putheader() (httplib.HTTPConnection method)
putp() (in module curses)
putrequest() (httplib.HTTPConnection method)
putsequences() (mhlib.Folder method)
putch() (in module msvcrt)
putwin() (curses.window method)
pwd
    module
pwd (module)
pwd() (ftplib.FTP method)
pwcurve() (in module gl)
py3kwarning (in module sys)
Py_AddPendingCall (C function)
Py_AddPendingCall()
Py_AtExit (C function)
Py_BEGIN_ALLOW_THREADS
    (C macro)
Py_BLOCK_THREADS (C macro)
Py_buffer (C type)
Py_buffer.buf (C member)
Py_buffer.internal (C member)
Py_buffer.itemsize (C member)
Py_buffer.ndim (C member)
Py_buffer.readonly (C member)

Python 3000
Python Editor
Python Enhancements
  PEP 100
  PEP 11
  PEP 205, [1
  PEP 207
Py_buffer.shape (C member)  PEP 208
Py_buffer.strides (C member)  PEP 217
Py_buffer.suboffsets (C member)  PEP 218, [1]
Py_BuildValue (C function)  PEP 227, [1]
Py_CLEAR (C function)  PEP 229
py_compile (module)  PEP 230, [1]
PY_COMPIL ED (in module imp)  PEP 230, [1]
Py_CompileString (C function)  PEP 232
Py_CompileString(), [1], [2]  PEP 234
Py_CompileStringFlags (C function)  PEP 236, [1]
Py_complex (C type)  PEP 237, [1]
Py_DECREF (C function)  PEP 238, [1]
Py_DECREF()  PEP 241
Py_END_ALLOW_THREADS  (C macro)  PEP 243
Py_END_OF_BUFFER (C variable)  PEP 249, [1]
Py_EndInterpreter (C function)  PEP 252, [1]
Py_EnterRecursiveCall (C function)  PEP 253, [1]
Py_eval_input (C variable)  PEP 255, [1]
Py_Exit (C function)  PEP 261, [1]
Py_False (C variable)  PEP 263, [1]
Py_FatalError (C function)  PEP 264
Py_FatalError()  PEP 266
Py_FdIsInteractive (C function)  PEP 273, [1]
Py_file_input (C variable)  PEP 275
Py_Finalize (C function)  PEP 277
Py_Finalize(), [1], [2], [3], [4]  PEP 278, [1]
Py_FindMethod (C function)  PEP 279
PY_FROZEN (in module imp)  PEP 282, [1]
Py_GetBuildInfo (C function)  PEP 285, [1]
Py_GetCompiler (C function)  PEP 288
Py_GetCopyright (C function)  PEP 289, [1]
Py_GetExecPrefix (C function)  PEP 292, [1]
Py_GetExecPrefix()  PEP 293
Py_GetPath (C function)  PEP 293
Py_GetPath(), [1]  PEP 3000
Py_GetPlatform (C function)  PEP 301, [1]
Py_GetPrefix (C function)  PEP 302, [1]
Py_GetPrefix()
Py_GetProgramFullPath (C function)
Py_GetProgramFullPath()
Py_GetProgramName (C function)
Py_GetPythonHome (C function)
Py_GetVersion (C function)
Py_INCREF (C function)
Py_INCREF()
Py_Initialize (C function)
Py_Initialize(), [1], [2], [3]
Py_InitializeEx (C function)
Py_InitModule (C function)
Py_InitModule3 (C function)
Py_InitModule4 (C function)
Py_IsInitialized (C function)
Py_IsInitialized()
Py_LeaveRecursiveCall (C function)
Py_Main (C function)
Py_NewInterpreter (C function)
Py_None (C variable)
py_object (class in ctypes)
Py_PRINT_RAW
Py_REFCNT (C macro)
Py_RETURN_FALSE (C macro)
Py_RETURN_NONE (C macro)
Py_RETURN_TRUE (C macro)
Py_SetProgramName (C function)
Py_SetProgramName(), [1], [2], [3]
Py_SetPythonHome (C function)
Py_single_input (C variable)
Py_SIZE (C macro)
PY_SOURCE (in module imp)
PY_SSIZE_T_MAX
py_suffix_importer() (in module imputil)
Py_TPFLAGS_BASETYPE (built-in variable)
Py_TPFLAGS_CHECKTYPES (built-in variable)
Py_TPFLAGS_DEFAULT (built-in variable)
Py_TPFLAGS_GC (built-in variable)
Py_TPFLAGS_HAVE_CLASS (built-in variable)
Py_TPFLAGS_HAVE_GC (built-in variable)

[9], [10], [11]
[17], [18], [1]
[25], [26], [2]
PEP 305, [1]
PEP 307, [1]
PEP 308, [1]
PEP 309
PEP 3100
PEP 3101, [
PEP 3105, [
PEP 3106
PEP 3110
PEP 3112, [
PEP 3116, [
PEP 3118
PEP 3119, [
PEP 3121
PEP 3127
PEP 3129
PEP 3137
PEP 314, [1]
PEP 3141, [
PEP 318, [1]
PEP 322, [1]
PEP 324, [1]
PEP 325
PEP 327
PEP 328, [1]
PEP 331
PEP 333, [1]
[9], [10], [11]
PEP 338, [1]
PEP 339
PEP 341
Py_TPFLAGS_HAVE_GETCHARBUFFER (built-in variable), [1]
Py_TPFLAGS_HAVE_INPLACEOPS (built-in variable)
Py_TPFLAGS_HAVE_ITER (built-in variable)
Py_TPFLAGS_HAVE_RICHCOMPARE (built-in variable)
Py_TPFLAGS_HAVE_SEQUENCE_IN (built-in variable)
Py_TPFLAGS_HAVE_WEAKREFS (built-in variable)
Py_TPFLAGS_HEAPTYPE (built-in variable)
Py_TPFLAGS_READY (built-in variable)
Py_TPFLAGS_READYING (built-in variable)
Py_tracefunc (C type)
Py_True (C variable)
Py_TYPE (C macro)
Py_UNBLOCK_THREADS (C macro)
Py_UNICODE (C type)
Py_UNICODE_ISALNUM (C function)
Py_UNICODE_ISALPHA (C function)
Py_UNICODE_ISDECIMAL (C function)
Py_UNICODE_ISDIGIT (C function)
Py_UNICODE_ISLINEBREAK (C function)
Py_UNICODE_ISLOWER (C function)
Py_UNICODE_ISSPACE (C function)
Py_UNICODE_ISTITLE (C function)
Py_UNICODE_ISUPPER (C function)
Py_UNICODE_TODECIMAL (C function)
Py_UNICODE_TODIGIT (C function)
Py_UNICODE_TOTITLE (C function)
Py_UNICODE_TOUNDER (C function)
Py_VaBuildValue (C function)
Py_VISIT (C function)
Py_XDECREF (C function)
Py_XDECREF()
PyCallIter_Check (C function)
PyCallIter_New (C function)
PyCallIter_Type (C variable)
PyCapsule (C type)
PyCapsule_CheckExact (C function)
PyCapsule_Destroy (C type)
PyCapsule_GetContext (C function)
PyCapsule_GetDestructor (C function)
PyCapsule_GetName (C function)
PyCapsule_GetPointer (C function)
PyCapsule_Import (C function)
PyCapsule_IsValid (C function)
PyCapsule_New (C function)
PyCapsule_SetContext (C function)
PyCapsule_SetDestructor (C function)
PyCapsule_SetName (C function)
PyCapsule_SetPointer (C function)
PyCell_Check (C function)
PyCell_GET (C function)
PyCell_Get (C function)
PyCell_New (C function)
PyCell_SET (C function)
PyCell_Set (C function)
PyCell_Type (C variable)
PyCellObject (C type)
PyCFunction (C type)
PyClass_Check (C function)
PyClass_IsSubclass (C function)
PyClass_Type (C variable)
PyClassObject (C type)
pyclbr (module)
PyCObject (C type)
PyCObject_AsVoidPtr (C function)
PyCObject_Check (C function)
PyCObject_FromVoidPtr (C function)
PyCObject_FromVoidPtrAndDesc (C function)
PyCObject_GetDesc (C function)
PyCObject_SetVoidPtr (C function)
PyCode_Check (C function)
PyCode_GetNumFree (C function)
PyCode_New (C function)
PyCode_NewEmpty (C function)
PyCode_Type (C variable)
PyCodec_BackslashReplaceErrors (C function)
PyCodec_Decode (C function)
PyCodec_Decoder (C function)
PyCodec_Encode (C function)
PyCodec_ENCODER (C function)
PyCodec_Encode (C function)
PyCodec_Encoder (C function)
PyCodec_IgnoreErrors (C function)
PyCodec_IncrementalDecoder (C function)
PyCodec_IncrementalEncoder (C function)
PyCodec_KnownEncoding (C function)
PyCodec_LookupError (C function)
PyCodec_Register (C function)
PyCodec_RegisterError (C function)
PyCodec_ReplaceErrors (C function)
PyCodecStreamReader (C function)
PyCodec_StreamWriter (C function)
PyCodec_StrictErrors (C function)
PyCodec_XMLCharRefReplaceErrors (C function)
PyCodeObject (C type)
PyCompileError
PyCompilerFlags (C type)
PyComplex_AsCComplex (C function)
PyComplex_Check (C function)
PyComplex_CheckExact (C function)
PyComplex_FromCComplex (C function)
PyComplex_FromDoubles (C function)
PyComplex_ImagAsDouble (C function)
PyComplex_RealAsDouble (C function)
PyComplex_Type (C variable)
PyComplexObject (C type)
PyDate_Check (C function)
PyDate_CheckExact (C function)
PyDateFromDate (C function)
PyDateTime_Check (C function)
PyDateTime_CheckExact (C function)
PyDateTime_DATE_GET_HOUR (C function)    PyTypeObject.t
PyDateTime_DATE_GET_MICROSECOND (C function)    PyTypeObject.t
PyDateTime_DATE_GET_MINUTE (C function)    PyTZInfo_Check
PyDateTime_DATE_GET_SECOND (C function)    PyUnicode_As
PyDateTime_FromDateAndTime (C function)    PyUnicode_As
PyDateTime_FromTimestamp (C function)    PyUnicode_As
PyDateTime_GET_DAY (C function)    PyUnicode_As
PyDateTime_GET_MONTH (C function)    PyUnicode_As
PyDateTime_GET_YEAR (C function)    PyUnicode_As
PyDateTime.TIME_GET_HOUR (C function)    PyUnicode_As
PyDateTime.TIME_GET_MICROSECOND (C function)    PyUnicode_As
PyDateTime.TIME_GET_MINUTE (C function)    PyUnicode_As
PyDateTime.TIME_GET_SECOND (C function)    PyUnicode_As
PyDelta_Check (C function)    PyUnicode_As
PyDelta_CheckExact (C function)    PyUnicode_As
PyDescr_IsData (C function)    PyUnicode_As
PyDescr_NewClassMethod (C function)    PyUnicode_As
PyDescr_NewGetSet (C function)    PyUnicode_As
PyDescr_NewMember (C function)    PyUnicode_As
PyDescr_NewMethod (C function)    PyUnicode_As
PyDescr_NewWrapper (C function)    PyUnicode_As
PyDict_Check (C function)    PyUnicode_As
PyDict_CheckExact (C function)    PyUnicode_As
PyDict_Clear (C function)    PyUnicode_As
PyDict_Contains (C function)    PyUnicode_As
PyDict_Copy (C function)    PyUnicode_As
PyDict_DelItem (C function)    PyUnicode_As
PyDict_DelItemString (C function)    PyUnicode_As
PyDict_GetItem (C function)    PyUnicode_As
PyDict_GetItemString (C function)    PyUnicode_As
PyDict_Items (C function)    PyUnicode_As
PyDict.Keys (C function)    PyUnicode_As
PyDict.Merge (C function)    PyUnicode_As
PyDict.MergeFromSeq2 (C function)    PyUnicode_As
PyDict.New (C function)    PyUnicode_As
PyDict.Next (C function)    PyUnicode_As
PyDict_SetItem (C function) PyUnicode_DecodeUTF8Stateful (C function)
PyDict_SetItemString (C function) PyDict_Type (C variable) PyDict_Size (C function)
PyDict_Type (C function) PyDict_Type (C variable) PyDict_Update (C function)
PyDict_Update (C function) PyDict_Update (C function) PyDict_Update (C function)
PyDict_Values (C function) PyDict_Values (C function) PyDict_Values (C function)
PyDictObject (C type) PyDictObject (C type) PyDictObject (C type)
PyDictProxy_New (C function) PyDictProxy_New (C function) PyDictProxy_New (C function)
PyDLL (class in ctypes) PyDLL (class in ctypes) PyDLL (class in ctypes)
pydoc (module) Pydoc (module) Pydoc (module)
PyErr_BadArgument (C function) PyErr_BadArgument (C function) PyErr_BadArgument (C function)
PyErr_BadInternalCall (C function) PyErr_BadInternalCall (C function) PyErr_BadInternalCall (C function)
PyErr_CheckSignals (C function) PyErr_CheckSignals (C function) PyErr_CheckSignals (C function)
PyErr_Clear (C function) PyErr_Clear (C function) PyErr_Clear (C function)
PyErr_ExceptionMatches (C function) PyErr_ExceptionMatches (C function) PyErr_ExceptionMatches (C function)
PyErr_ExceptionMatches() PyErr_ExceptionMatches() PyErr_ExceptionMatches()
PyErr_Fetch (C function) PyErr_Fetch (C function) PyErr_Fetch (C function)
PyErr_Fetch() PyErr_Fetch() PyErr_Fetch()
PyErr_Format (C function) PyErr_Format (C function) PyErr_Format (C function)
PyErr_GivenExceptionMatches (C function) PyErr_GivenExceptionMatches (C function) PyErr_GivenExceptionMatches (C function)
PyErr_NewException (C function) PyErr_NewException (C function) PyErr_NewException (C function)
PyErr_NewExceptionWithDoc (C function) PyErr_NewExceptionWithDoc (C function) PyErr_NewExceptionWithDoc (C function)
PyErr_NoMemory (C function) PyErr_NoMemory (C function) PyErr_NoMemory (C function)
PyErr_NormalizeException (C function) PyErr_NormalizeException (C function) PyErr_NormalizeException (C function)
PyErr_Occurred (C function) PyErr_Occurred (C function) PyErr_Occurred (C function)
PyErr_Occurred() PyErr_Occurred() PyErr_Occurred()
PyErr_Print (C function) PyErr_Print (C function) PyErr_Print (C function)
PyErr_PrintEx (C function) PyErr_PrintEx (C function) PyErr_PrintEx (C function)
PyErr_Restore (C function) PyErr_Restore (C function) PyErr_Restore (C function)
PyErr_Restore() PyErr_Restore() PyErr_Restore()
PyErr_SetExcFromWindowsErr (C function) PyErr_SetExcFromWindowsErrWithFilename (C function) PyErr_SetExcFromWindowsErrWithFilenameObject (C function)
PyErr_SetExcFromWindowsErrWithFilenameObject (C function) PyErr_SetExcFromWindowsErrWithFilenameObject (C function) PyErr_SetExcFromWindowsErrWithFilenameObject (C function)
PyErr_SetFromErrno (C function) PyErr_SetFromErrno (C function) PyErr_SetFromErrno (C function)
PyErr_SetFromErrnoWithFilename (C function) PyErr_SetFromErrnoWithFilename (C function) PyErr_SetFromErrnoWithFilename (C function)
PyErr_SetFromErrnoWithFilenameObject (C function) PyErr_SetFromErrnoWithFilenameObject (C function) PyErr_SetFromErrnoWithFilenameObject (C function)
PyEval_SaveThread (C function)
PyEval_SaveThread(), [1]
PyEval_SetProfile (C function)
PyEval_SetTrace (C function)
PyEval_ThreadsInitialized (C function)
PyExc_ArithmeticError
PyExc_AssertionError
PyExc_AttributeError
PyExc_BaseException
PyExc_EnvironmentError
PyExc_EOFError
PyExc_OSError
PyExc_ReferenceError
PyExc_SyntaxError
PyExc_TypeError
PyExc_ValueError
PyExc_ZeroDivisionError
PyUnicodeEncodeError_SetStart (C function)
PyUnicodeObject (C type)
PyUnicodeTranslateError_Create (C function)
PyUnicodeTranslateError_GetEnd (C function)
PyUnicodeTranslateError_GetObject (C function)
PyUnicodeTranslateError_GetReason (C function)
PyUnicodeTranslateError_GetStart (C function)
PyUnicodeTranslateError_SetEnd (C function)
PyUnicodeTranslateError_SetReason (C function)
PyUnicodeTranslateError_SetStart (C function)
PyVarObject (C type)
PyVarObject.ob_size (C member)
PyVarObject_HEAD_INIT (C macro)
PyWeakref_Check (C function)
PyWeakref_CheckProxy (C function)
PyWeakref_CheckRef (C function)
PyWeakref_GET_OBJECT (C function)
PyWeakref_GetObject (C function)
PyWeakref_NewProxy (C function)
PyWeakref_NewRef (C function)
PyWrapper_New (C function)
PyZipFile (class in zipfile)
Index – Q

qdevice() (in module fl)
QDPoint (class in aetypes)
QDRectangle (class in aetypes)
qenter() (in module fl)
qiflush() (in module curses)
QName (class in xml.etree.ElementTree)
qread() (in module fl)
qreset() (in module fl)
qsize() (multiprocessing.Queue method)
    (Queue.Queue method)
qtest() (in module fl)
quantize() (decimal.Context method)
    (decimal.Decimal method)
quopri (module)
quote() (in module email.utils)
    (in module pipes)
    (in module rfc822)
    (in module urllib)
QUOTE_ALL (in module csv)
QUOTE_MINIMAL (in module csv)
QUOTE_NONE (in module csv)
QUOTE_NONNUMERIC (in module csv)
quote_plus() (in module urllib)
quoteattr() (in module xml.sax.saxutils)
quotechar (csv.Dialect attribute)
quoting (csv.Dialect attribute)
(class in Queue)
(module)
queue (sched.scheduler attribute)
Index – R

r_eval() (rexec.RExec method)

r_exec() (rexec.RExec method)

r_execfile() (rexec.RExec method)

r_import() (rexec.RExec method)

R_OK (in module os)

r_open() (rexec.RExec method)

r_reload() (rexec.RExec method)

r_unload() (rexec.RExec method)

radians() (in module math)

(radians() (in module turtle)

RadioButtonGroup (class in msilib)

radiogroup() (msilib.Dialog method)

radix() (decimal.Context method)

(decimal.Decimal method)

RADIXCHAR (in module locale)

raise

statement, [1]

raise (2to3 fixer)

raise an exception

RAISE_VARARGS (opcode)

raising

exception

RAND_add() (in module ssl)

RAND_egd() (in module ssl)

RAND_status() (in module ssl)

randint() (in module random)

random (module)

random() (in module random)

randrange() (in module random)

range

built-in function

Range (class in aetypes)

range() (built-in function)

ratecv() (in module audioop)

ratio() (difflib.SequenceMatcher method)

renames() (in module os)

reorganize() (in module gdbm)

repeat() (in module operator)

(repeat() (in module timeit)

(repeat() (timeit.Timer method)

(repetition

operation

replace() (curses panel)

replace() (datetime.datetime method)

replace() (datetime.timedelta method)

replace() (datetime.timedelta method)

replace() (modulefinder.ModuleFinder method)

replace() (modulefinder.replace_child)

replace() (sgmllib.replace_whitespace)

replace() (str method)

replace_errors() (in module codecs)

replace_header() (email.message.Message method)

replace_history_item() (in module readline)

replace_whitespace (textwrap.TextWrapper method)

replaceChild() (xml.dom.Node method)

ReplacePackage() (in module modulefinder)

report() (filecmp.cmp)

(modulefinder)

REPORT_CDIFF

report_failure() (in module modulefinder)

REPORT_NDIFF

REPORT_ONLY

report_exception() (in module modulefinder)

report_partial_closure() (in module modulefinder)

report() (str method)

REPORT_UDIFF

report_unbalanced() (sgmllib.SGMLParser method)

report_unexpected_exception() (in module modulefinder)
Rational (class in numbers)
raw (io.BufferedIOBase attribute)
raw input
raw string
raw() (in module curses)
raw_decode() (json.JSONDecoder method)
raw_input
  built-in function, [1]
 raw_input (2to3 fixer)
 raw_input() (built-in function)
    (code.InteractiveConsole method)
 RawArray() (in module multiprocessing.sharedctypes)
 RawConfigParser (class in ConfigParser)
 RawDescriptionHelpFormatter (class in argparse)
 RawIOBase (class in io)
 RawPen (class in turtle)
 RawTextHelpFormatter (class in argparse)
 RawTurtle (class in turtle)
 RawValue() (in module multiprocessing.sharedctypes)
 RBRACE (in module token)
 re
  module, [1], [2]
 re (module)
    (re.MatchObject attribute)
 read() (array.array method)
    (ConfigParser.RawConfigParser method)
    (bz2.BZ2File method)
    (chunk.Chunk method)
    (codecs.StreamReader method)
    (file method)
    (httpbin.HTTPResponse method)
    (imaplib.IMAP4 method)
    (in module imgfile)
    (in module os)
    (io.BufferedIOBase method)
    (io.BufferedReader method)
    (io.RawIOBase method)
(wave.Wave_read method)
readinto() (io.BufferedIOBase method)
    (io.RawIOBase method)
readline
    module
readline (module)
readline() (bz2.BZ2File method)
    (codecs.StreamReader method)
    (distutils.text_file.TextFile method)
    (file method), [1]
    (imaplib.IMAP4 method)
    (io.IOBase method)
    (io.TextIOWrapper method)
    (mmap.mmap method)
    (multifile.MultiFile method)
readlines() (bz2.BZ2File method)
    (codecs.StreamReader method)
    (distutils.text_file.TextFile method)
    (file method)
    (io.IOBase method)
    (multifile.MultiFile method)
readlink() (in module os)
readmodule() (in module pycparser)
readmodule_ex() (in module pycparser)
READONLY
readonly (memoryview attribute)
readPlist() (in module plistlib)
readPlistFromResource() (in module plistlib)
readPlistFromString() (in module plistlib)
readscaled() (in module imgfile)
READY (in module cd)
ready() (multiprocessing.pool.AsyncResult method)
Real (class in numbers)
real (numbers.Complex attribute)
Real Media File Format
real_quick_ratio() (difflib.SequenceMatcher method)
realloc()
realpath() (in module os.path)  
reason (exceptions.UnicodeError attribute)  
   (httpplib.HTTPResponse attribute)  
   (ssl.SSLLError attribute)  
   (urllib2.HTTPError attribute)  
   (urllib2.URLError attribute)  
reattach() (ttk.Treeview method)  
rebinding  
   name  
reccontrols() (ossaudiodev.oss_mixer_device method)  
recent() (imaplib.IMAP4 method)  
rect() (in module cmath)  
rectangle() (in module curses.textpad)  
recursive  
   object  
recv() (asyncore.dispatcher method)  
   (multiprocessing.Connection method)  
   (socket.socket method)  
recv_bytes() (multiprocessing.Connection method)  
recv_bytes_into() (multiprocessing.Connection method)  
recv_into() (socket.socket method)  
recvfrom() (socket.socket method)  
recvfrom_into() (socket.socket method)  
redirect_request() (urllib2.HTTPRedirectHandler method)  
redisplay() (in module readline)  
redraw_form() (fl.form method)  
redrawln() (curses.window method)  
redrawwin() (curses.window method)  
reduce (2to3 fixer)  
reduce() (built-in function)  
   (in module functools)  
ref (class in weakref)  
reference  
   attribute  
RFC 1730  
RFC 1738  
RFC 1750  
RFC 1766, [1]  
RFC 1808, [1]  
RFC 1832, [1]  
RFC 1866  
RFC 1869, [1]  
RFC 1894  
RFC 2033  
RFC 2045, [1]  
RFC 2046, [1]  
RFC 2047, [1]  
RFC 2060, [1]  
RFC 2068  
RFC 2104, [1]  
RFC 2109, [1]  
RFC 2342  
RFC 2368  
RFC 2396, [1]  
RFC 2487  
RFC 2616, [1]  
RFC 2732, [1]  
RFC 2774  
RFC 2817  
RFC 2818  
RFC 2821  
RFC 2822, [11], [12], [13]  
RFC 2823, [21], [22], [23]  
RFC 2964
reference count
counting
ReferenceError, [1]
ReferenceType (in module weakref)
refresh() (mhlib.Folder method)
refresh() (curses.window method)
REG_BINARY (in module _winreg)
REG_DWORD (in module _winreg)
REG_DWORD_BIG_ENDIAN (in module _winreg)
REG_DWORD_LITTLE_ENDIAN (in module _winreg)
REG_EXPAND_SZ (in module _winreg)
REG_FULL_RESOURCE_DESCRIPTOR (in module _winreg)
REG_LINK (in module _winreg)
REG_MULTI_SZ (in module _winreg)
REG_NONE (in module _winreg)
REG_RESOURCE_LIST (in module _winreg)
REG_RESOURCE_REQUIREMENTS_LIST (in module _winreg)
REG_SZ (in module _winreg)
RegexObject (class in re)
register() (abc.ABCMeta method)
  (in module atexit)
  (in module codecs)
  (in module webbrower)
  (multiprocessing.managers.BaseManager
  method)
  (select.epoll method)
  (select.poll method)
register_adapter() (in module sqlite3)
register_archive_format() (in module shutil)
register_converter() (in module sqlite3)
register_dialect() (in module csv)
register_error() (in module codecs)
register_function()
(SimpleXMLRPCServer.CGIXMLRPCRequestHandler
  method)
  (SimpleXMLRPCServer.SimpleXMLRPCServer
  module)
rfc2109 (cookieli
rfc2109_as_nets
attribute)
rfc2965 (cookieli
rfc822
codecs)
rfc822 (module)
rfc822_escape()
method)
register_instance()
(SimpleXMLRPCServer.CGIXMLRPCRequestHandler
method)
(SimpleXMLRPCServer.SimpleXMLRPCServer
method)
register_introspection_functions()
(SimpleXMLRPCServer.CGIXMLRPCRequestHandler
method)
(SimpleXMLRPCServer.SimpleXMLRPCServer
method)
register_multicall_functions()
(SimpleXMLRPCServer.CGIXMLRPCRequestHandler
method)
(SimpleXMLRPCServer.SimpleXMLRPCServer
method)
register_namespace() (in module
xml.etree.ElementTree)
register_optionflag() (in module doctest)
register_shape() (in module turtle)
registerDOMImplementation() (in module xml.dom)
registerResult() (in module unittest)
relative
    URL
    import
release() (in module platform)
    (logging.Handler method)
    (multiprocessing.Lock method)
    (multiprocessing.RLock method)
    (thread.lock method)
    (threading.Condition method)
    (threading.Lock method)
    (threading.RLock method)
    (threading.Semaphore method)
release_lock() (in module imp)
reload
RFC_4122 (in module
rfile (BaseHTTP
attribute)
rfind() (in module
(mmap.mmap
(str method)
rgb_to_hls() (in r
rgb_to_hsv() (in r
rgb_to_yiq() (in r
RGBColor (class
right (filecmp.dircmp
right() (in module
right_list (filecmp:
right_only (filecmp
RIGHTSHIFT (in
RIGHTSHIFTEQ
rindex() (in modu
(str method)
rjust() (in module
(str method)
rlcompleter
    module
rlcompleter (modu
rlecode_hqx() (ir
rledecode_hqx() RLIM_INFINITY
RLIMIT_AS (in n
RLIMIT_CORE ( RLIMIT_CPU ( RLIMI
RLIMIT_DATA (i RLIMIT_FSIZE ( RLIMI
RLIMIT_MEML
RLIMIT_NOFILE RLIMIT_NPROC RLIMIT_OFI
RLIMIT_NPROC RLIMIT_VMEM (
built-in function, [1], [2], [3]
reload() (built-in function)
relpath() (in module os.path)
remainder() (decimal.Context method)
remainder_near() (decimal.Context method)
   (decimal.Decimal method)
remove() (array.array method)
   (collections.deque method)
   (in module os)
   (list method)
   (mailbox.MH method)
   (mailbox.Mailbox method)
   (set method)
   (xml.etree.ElementTree.Element method)
remove_flag() (mailbox.MaildirMessage method)
   (mailbox.MMDFMessage method)
   (mailbox.mboxMessage method)
remove_folder() (mailbox.Maildir method)
   (mailbox.MH method)
remove_history_item() (in module readline)
remove_label() (mailbox.BabylMessage method)
remove_option() (ConfigParser.RawConfigParser method)
   (optparse.OptionParser method)
remove_pyc() (msilib.Directory method)
remove_section() (ConfigParser.RawConfigParser method)
remove_sequence() (mailbox.MHMessage method)
remove_tree() (in module distutils.dir_util)
removeAttribute() (xml.dom.Element method)
removeAttributeNode() (xml.dom.Element method)
removeAttributeNS() (xml.dom.Element method)
removeChild() (xml.dom.Node method)
removedirs() (in module os)
removeFilter() (logging.Handler method)
   (logging.Logger method)
removeHandler() (in module unittest)

RLock (class in multiprocessing)
RLock() (in module threading)
rmd() (ftplib.FTP)
rmdir() (in module os)
RMFF
rms() (in module audioop)
rmtree() (in module shutil)
rnopen() (in module RO
RobotFileParser
robotparser (module robots.txt
rollback() (sqlite3)
ROT_FOUR (opcode)
ROT_THREE (opcode)
ROT_TWO (opcode)
rotate() (collections)
   (decimal.Context method)
   (decimal.Decimal method)
RotatingFileHandler
round() (built-in function)
Rounded (class in decimal)
Row (class in sqlite3)
row_factory (sqlite3.Connection attribute)
rowcount (sqlite3.Cursor attribute)
RPAR (in module token)
rpartition() (str method)
RSQB
ruuf (in module bsddb)
RPC (module)
rpc_paths (SimpleXMLRPCServer)
rsplit() (str method)
RSQB (in module simplexmlrpcserver)
rstrip() (in module string)
removemessages() (mhlib.Folder method)
removeResult() (in module unittest)
rename() (ftplib.FTP method)
    (imaplib.IMAP4 method)
    (in module os)
renames (2to3 fixer)

Run script
run() (bdb.Bdb method)
    (distutils.cmd.Command method)
    (doctest.DocTestRunner method)
    (hotshot.Profile method)
    (in module pdb)
    (in module profile)
    (multiprocessing.
pdb.Pdb method)
    (profile.Profile method)
    (sched.scheduler)
    (threading.Thread)
    (trace.Trace)
    (unittest.TestCase)
    (unittest.TestSuite)
    (wsgiref.headers)
run_docstring_examples() (in module doctest)
run_module() (in module runpy)
run_path() (in module runpy)
run_script() (modulefinder.ModuleFinder method)
run_setup() (in module distutils.core)
run_unittest() (in module test.test_support)
runcall() (bdb.Bdb method)
    (hotshot.Profile method)
    (in module pdb)
    (pdb.Pdb method)
    (profile.Profile method)
runcode() (code.InteractiveInterpreter method)
runcxt() (bdb.Bdb method)
    (hotshot.Profile method)
    (in module profile)
Index – S

S (in module re) showsyntaxerror() (code.InteractiveInterpreter method)
S_ENFMT (in module stat) showtraceback() (code.InteractiveInterpreter method)
s_eval() (rexec.RExec method) showturtle() (in module turtle)
s_exec() (rexec.RExec method) showwarning() (in module warnings)
s_execfile() (rexec.RExec method) shuffle() (in module random)
S_IEXEC (in module stat) shutdown() (in module SocketServer)
S_IFBLK (in module stat) (in module findertools)
S_IFCHR (in module stat) (in module logging)
S_IFDIR (in module stat) (multiprocessing.managers.BaseManager method)
S_IFIFO (in module stat) (socket.socket method)
S_IFLNK (in module stat) shutil (module)
S_IFMT() (in module stat) SIG_DFL (in module signal)
S_IFREG (in module stat) SIG_IGN (in module signal)
S_IFSOCK (in module stat) SIGINT, [1]
s_iread() (rexec.RExec method) siginterrupt() (in module signal)
S_IROTH (in module stat) module, [1]
S_IRUSR (in module stat) signal (module)
simport() (rexec.RExec method) signal() (in module simple
S_IRWXG (in module stat) statement
S_IRWXO (in module stat) Simple Mail Tra
S_IRWXU (in module stat) SimpleCookie (class	in	Cookie)
s_isblk() (in module stat) simplefilter() (in module warnings)
s_ischr() (in module stat) SimpleHandler
S_ISDIR() (in module stat) SimpleHTTPRequestHandler (class	in	SimpleHTTPServer)
s_isdir() (in module stat) SimpleHTTPServer
S_ISFIFO() (in module stat) module
S_ISGID (in module stat) SimpleHTTPSe
S_ISLNK() (in module stat) SimpleHTTPSe module
S_ISREG() (in module stat) SimpleXMLRPC
S_ISSOCK() (in module stat)
S_IWUSR (in module stat)
S_IXGRP (in module stat)
S_IXOTH (in module stat)
S_IXUSR (in module stat)
s_reload() (rexec.RExec method)
s_unload() (rexec.RExec method)
safe_substitute() (string.Template method)
SafeConfigParser (class in ConfigParser)
saferepr() (in module pprint)
same_files (filecmp.dircmp attribute)
same_quantum() (decimal.Context method)
    (decimal.Decimal method)
samefile() (in module os.path)
sameopenfile() (in module os.path)
samestat() (in module os.path)
sample() (in module random)
save() (cookielib.FileCookieJar method)
save_bgn() (httplib.HTMLParser method)
save_end() (httplib.HTMLParser method)
SaveKey() (in module _winreg)
savetty() (in module curses)
SAX2DOM (class in xml.dom.pulldom)
SAXException
SAXNotRecognizedException
SAXNotSupportedException
SAXParseException
scale() (in module imageop)
scaleb() (decimal.Context method)
    (decimal.Decimal method)
scalebarvalues() (FrameWork.ScrolledWindow method)
scanf()
sched (module)
scheduler (class in sched)
schema (in module msilib)
sci() (in module fpformat)
scope, [1]
Scrap Manager
SimpleXMLRPCServer
SimpleXMLRPCServer
SimpleXMLRPCServer
    (module)
sin() (in module
    (in module r
singleton
tuple
sinh() (in module
    (in module r
site (module)
    site command line
        --user-base
        --user-site
    site-packages
directory
    site-python
directory
sitecustomize
    module
    size (struct.Struct
        (tarfile.TarIn
    size() (ftplib.FTP
        (mmap.mmap
    Sized (class in
    sizeof() (in mod
    SKIP (in module
    skip() (chunk.C
        (in module u
    skipIf() (in mod
    skipinitialspace (csv.Dialect attribute)
    skipped (unittest
    skippedEntity() (xml.sax.handler.ContentHandler method)
    SkipTest
    skipTest() (unittest
    skipUnless() (in
Screen (class in turtle)
screensize() (in module turtle)
script_from_examples() (in module doctest)
scroll() (curses.window method)
scrollbar_callback() (FrameWork.ScrolledWindow method)
scrollbars() (FrameWork.ScrolledWindow method)
ScrolledCanvas (class in turtle)
ScrolledText (module)
scrollok() (curses.window method)
search
  path, module, [1], [2], [3], [4], [5], [6]
search() (imaplib.IMAP4 method)
  (in module re)
  (re.RegexObject method)
SEARCH_ERROR (in module imp)
second (datetime.datetime attribute)
  (datetime.time attribute)
section_divider() (multifile.MultiFile method)
sections() (ConfigParser.RawConfigParser method)
secure (cookie librarian.Cookie attribute)
Secure Hash Algorithm
secure hash algorithm, SHA1, SHA224, SHA256, SHA384, SHA512
Secure Sockets Layer
security
  CGI
see() (ttk.Treeview method)
seed() (in module random)
seek() (bz2.BZ2File method)
  (chunk.Chunk method)
  (file method)
  (io.IOBase method)
  (io.TextIOBase method)
  (mmap.mmap method)
  (multifile.MultiFile method)
SEEK_CUR (in module os)
SLASH (in module token)
SLASHEQUAL (in module token)
sleep() (ntplib.ntpmon module)
slice() (FrameWork.ScrolledWindow method)
  slice, [1]
  assignment
  built-in function
  object
  operation
slice (built-in class)
SLICE+0 (opcode)
SLICE+1 (opcode)
SLICE+2 (opcode)
SLICE+3 (opcode)
slice (in module inspect)
slicing, [1], [2]
  assignment
  extended
SmartCookie (class in Cookie)
SMTP
  protocol
SMTP (class in smtplib)
SMTP_SSL (class in smtplib)
SMTPAuthenticationError
SMTPConnectError
SMTPDataError
SMTPException
SMTPHandler (class in logging.handlers)
SMTPHeloError
smtplib (module)
smtplib (module)
smtplib (module)
smtplib (module)
SMTPTCPHostError
SMTPUDPHostError
SMTPPortError
SMTPHandler (class in logging.handlers)
SMTPMSMTPErrors
SMTPServer (class in smtplib)
SMTPServerDisconnected
SND_ALIAS (in mod...
SEEK_END (in module os)
SEEK_SET (in module os)
seekable() (io.IOBase method)
segcountproc (C type)
Select (class in Tix)
select (module)
select() (imaplib.IMAP4 method)
selected_alpn_protocol() (ssl.SSLSocket method)
selected_npn_protocol() (ssl.SSLSocket method)
selection() (ttk.Treeview method)
selection_add() (ttk.Treeview method)
selection_remove() (ttk.Treeview method)
selection_set() (ttk.Treeview method)
selection_toggle() (ttk.Treeview method)
Semaphore (class in multiprocessing)
Semaphore()
(selected in multiprocessing.managers.SyncManager method)
semaphores, binary
SEMI (in module token)
send() (aetools.TalkTo method)
send_bytes() (multiprocessing.Connection method)
send_error()
send_flowing_data () (formatter.writer method)
(send_header ()
(BaseHTTPServer.BaseHTTPRequestHandler method)
send_hor_rule () (formatter.writer method)
send_label_data () (formatter.writer method)
send_line_break () (formatter.writer method)
send_paragraph () (formatter.writer method)
(send_response ()
(BaseHTTPServer.BaseHTTPRequestHandler method)
(send_signal () (subprocess.Popen method)
(sendall () (socket.socket method)
(sendcmd () (ftplib.FTP method)
(sendfile () (wsgiref.handlers.BaseHandler method)
(sendmail () (smtplib.SMTP method)
(sendto () (socket.socket method)
sep (in module os)
Separator () (in module FrameWork)
sequence
  item
  iteration
  object, [1], [2], [3], [4], [5], [6], [7], [8]
  types, mutable
  types, operations on, [1]
Sequence (class in collections)
sequence (in module msilib)
sequence2st () (in module parser)
sequenceIncludes () (in module operator)
SequenceMatcher (class in difflib), [1]
SerialCookie (class in Cookie)
serializing
  objects
serve_forever () (SocketServer.BaseServer method)
server

(source character set)
sourcehook () (subprocess.Popen method)
span () (re.MatchObject method)
spawn () (distutils.ccompiler.CCompiler method)
(spawnl () (in module pty)
spawnle () (in module os)
spawnlpe () (in module os)
spawnnv () (in module os)
spawnne () (in module os)
spawnnp () (in module os)
spawnnvpe () (in module os)
special
  attribute
  attribute, generic
  specified_attributes
Sequence (class in collections)
Sequence (in module msilib)
Sequence2st () (in module parser)
sequenceIncludes () (in module operator)
SequenceMatcher (class in difflib), [1]
SerialCookie (class in Cookie)
serializing
  objects
serve_forever () (SocketServer.BaseServer method)
server

(ossaudiodev.oss_audio_device method)
splash () (in module MacOS)
split () (in module os)
split () (in module re)
split () (in module shlex)
split () (in module string)
split () (re.RegexObject method)
split () (str method)
split_quoted () (in module distutils.util)
splitdrive () (in module os)
splitext () (in module os)
splitfields () (in module string)
splitlines () (str method)
(unicode method)
WWW, [1]
server (BaseHTTPServer.BaseHTTPRequestHandler attribute)
server_activate() (SocketServer.BaseServer method)
server_address (SocketServer.BaseServer attribute)
server_bind() (SocketServer.BaseServer method)
server_close() (SocketServer.BaseServer method)
server_software (wsgiref.handlers.BaseHandler attribute)
server_version (BaseHTTPServer.BaseHTTPRequestHandler attribute)
   (SimpleHTTPServer.SimpleHTTPRequestHandler attribute)
ServerProxy (class in xmlrpclib)
session_stats() (ssl.SSLContext method)
set
display
   object, [1], [2], [3]
set (built-in class)
Set (class in collections)
   (class in sets)
Set Breakpoint
set type
   object
set() (ConfigParser.RawConfigParser method)
   (ConfigParser.SafeConfigParser method)
   (Cookie.Morsel method)
   (EasyDialogs.ProgressBar method)
   (ossaudiodev.oss_mixer_device method)
   (test.test_support.EnvironmentVarGuard method)
   (threading.Event method)
   (ttk.Combobox method)
   (ttk.Treeview method)
   (xml.etree.ElementTree.Element method)
set_all()
set_allowed_domains() (cookielib.DefaultCookiePolicy
set_alpn_protocols() (ssl.SSLContext method)
set_app() (wsgiref.simple_server.WSGIServer method)
set_authorizer() (sqlite3.Connection method)
set_blocked_domains() (cookielib.DefaultCookiePolicy method)
set_boundary() (email.message.Message method)
set_break() (bdb.Bdb method)
set_charset() (email.message.Message method)
set_children() (ttk.Treeview method)
set_ciphers() (ssl.SSLContext method)
set_completer() (in module readline)
set_completer_delims() (in module readline)
set_completion_display_matches_hook() (in module readline)
set_continue() (bdb.Bdb method)
set_conversion_mode() (in module ctypes)
set_cookie() (cookielib.CookieJar method)
set_cookie_if_ok() (cookielib.CookieJar method)
set_current() (msilib.Feature method)
set_date() (mailbox.MaildirMessage method)
set_debug() (in module gc)
set_debuglevel() (ftplib.FTP method)
(set_connection) (httplib.HTTPConnection method)
(nntplib.NNTP method)
(poplib.POP3 method)
(smtplib.SMTP method)
(telnetlib.Telnet method)
set_default_type() (email.message.Message method)
set_default_verify_paths() (ssl.SSLContext method)
set_defaults() (argparse.ArgumentParser method)
(starmap() (in module itertools)
(start() (hotshot.Profile method)
(multiprocessing. Process method)
(re.MatchObject)
(threading.Thread)
(xml.etree.ElementTree)
(set_color() (in curses)
(start_component()) (msilib.Directory method)
(start_new_thread()) (in module threading)
(startbody() (MimeWriter.MimeWriter method)
(StartCdataSectionHandler)
set_flags() (mailbox.MaildirMessage method)
   (mailbox.MMDFMessage method)
   (mailbox.mboxMessage method)
set_form_position() (fl.form method)
set_from() (mailbox.mboxMessage method)
   (mailbox.MMDFMessage method)
set_graphics_mode() (in module fl)
set_history_length() (in module readline)
set_include_dirs() (distutils.ccompiler.CCompiler method)
set_info() (mailbox.MaildirMessage method)
set_labels() (mailbox.BabylMessage method)
set_last_error() (in module ctypes)
set_link_objects() (distutils.ccompiler.CCompiler method)
set_library_dirs() (distutils.ccompiler.CCompiler method)
SET_LINENO (opcode)
set_link_objects() (distutils.ccompiler.CCompiler method)
setliteral (2to3 fixer)
set_location() (bsddb.bsddbobject method)
set_next() (bdb.Bdb method)
set_nonstandard_attr() (cookielib.Cookie method)
set_npn_protocols() (ssl.SSLContext method)
set_ok() (cookielib.CookiePolicy method)
set_option_negotiation_callback() (telnetlib.Telnet method)
set_output_charset() (gettext.NullTranslations method)
set_param() (email.message.Message method)
set_pasv() (ftplib.FTP method)
set_payload() (email.message.Message method)
set_policy() (cookielib.CookieJar method)
set_position() (xdrlib.Unpacker method)
set_pre_input_hook() (in module readline)
set_progress_handler() (sqlite3.Connection method)
set_proxy() (urllib2.Request method)
set_python_build() (in module distutils.sysconfig)
set_quit() (bdb.Bdb method)

(xml.parsers.expat.xmlparser method)
StartDoctypeDeclHandler()
(xml.parsers.expat.xmlparser method)
startDocument()
(xml.sax.handler.ContentHandler method)
StartElement()
(xml.sax.handler.ContentHandler method)
StartElementNS()
(xml.sax.handler.ContentHandler method)
STARTF_USESHOWWINDOW (in module subprocess)
STARTF_USESTDHANDLES (in module subprocess)
startfile() (in module startmultipartbo)
startfile() (in module subprocess)
startTest() (unittest.TestResult method)
startTestRun() (unittest.TestResult method)
starttls() (smtplib STARTUPINFO)
stat
   module
stat(stat(module))
stat(stat(module))
nntplib.__init__.POP
stat_float_times() (os)
state() (ttk.Widget)

* [1]
** [1]
set_recsrc() (ossaudiodev.oss_mixer_device method)
set_return() (bdb.Bdb method)
set_runtime_library_dirs()
(distutils.ccompiler.CCompiler method)
set_seq1() (difflib.SequenceMatcher method)
set_seq2() (difflib.SequenceMatcher method)
set_seqs() (difflib.SequenceMatcher method)
set_sequences() (mailbox.MH method)
  (mailbox.MHMessage method)
set_server_documentation()
(DocXMLRPCServer.DocCGIXMLRPCRequestHandler method)
  (DocXMLRPCServer.DocXMLRPCServer method)
set_server_name()
(DocXMLRPCServer.DocCGIXMLRPCRequestHandler method)
  (DocXMLRPCServer.DocXMLRPCServer method)
set_server_title()
(DocXMLRPCServer.DocCGIXMLRPCRequestHandler method)
  (DocXMLRPCServer.DocXMLRPCServer method)
set_servername_callback() (ssl.SSLContext method)
set_spacing() (formatter.formatter method)
set_startup_hook() (in module readline)
set_step() (bdb.Bdb method)
set_subdir() (mailbox.MaildirMessage method)
set_terminator() (asynchat.async_chat method)
set_threshold() (in module gc)
set_trace() (bdb.Bdb method)
  (in module bdb)
  (in module pdb)
  (pdb.Pdb method)
set_tunnel() (httplib.HTTPConnection method)
set_type() (email.message.Message method)
set_unittest_reportflags() (in module doctest)
set_unixfrom() (email.message.Message method)
set_until() (bdb.Bdb method)
set_url() (robotparser.RobotFileParser method)
set_usage() (optparse.OptionParser method)
set_userptr() (curses.panel.Panel method)
set_visible() (mailbox.BabylMessage method)
set_wakeup_fd() (in module signal)
setacl() (imaplib.IMAP4 method)
setannotation() (imaplib.IMAP4 method)
setarrowcursor() (in module FrameWork)
setattr() (built-in function)
setAttribute() (xml.dom.Element method)
setAttributeNode() (xml.dom.Element method)
setAttributeNodeNS() (xml.dom.Element method)
setAttributeNS() (xml.dom.Element method)
SetBase() (xml.parsers.expat.xmlparser method)
setblocking() (socket.socket method)
setByteStream() (xml.sax.xmlreader.InputSource method)
setcbreak() (in module tty)
setCharacterStream() (xml.sax.xmlreader.InputSource method)
setcheckinterval() (in module sys), [1]
setcomptype() (aifc.aifc method)
    (sunau.AU_write method)
    (wave.Wave_write method)
setContentHandler() (xml.sax.xmlreader.XMLReader method)
setcontext() (in module decimal)
    (mhlib.MH method)
SetCreatorAndType() (in module MacOS)
setCurrent() (mhlib.Folder method)
setDaemon() (threading.Thread method)
setdefault() (dict method)
setdefaultencoding() (in module sys)
setdefaulttimeout() (in module socket)
setdlopenflags() (in module sys)
setDocumentLocator()
    (xml.sax.handler.ContentHandler method)
setDTDHandler() (xml.sax.xmlreader.XMLReader method)
set_usage() (optparse.OptionParser method)
set_userptr() (curses.panel.Panel method)
set_visible() (mailbox.BabylMessage method)
set_wakeup_fd() (in module signal)
setacl() (imaplib.IMAP4 method)
setannotation() (imaplib.IMAP4 method)
setarrowcursor() (in module FrameWork)
setattr() (built-in function)
setAttribute() (xml.dom.Element method)
setAttributeNode() (xml.dom.Element method)
setAttributeNodeNS() (xml.dom.Element method)
setAttributeNS() (xml.dom.Element method)
SetBase() (xml.parsers.expat.xmlparser method)
setblocking() (socket.socket method)
setByteStream() (xml.sax.xmlreader.InputSource method)
setcbreak() (in module tty)
setCharacterStream() (xml.sax.xmlreader.InputSource method)
setcheckinterval() (in module sys), [1]
setcomptype() (aifc.aifc method)
    (sunau.AU_write method)
    (wave.Wave_write method)
setContentHandler() (xml.sax.xmlreader.XMLReader method)
setcontext() (in module decimal)
    (mhlib.MH method)
SetCreatorAndType() (in module MacOS)
setCurrent() (mhlib.Folder method)
setDaemon() (threading.Thread method)
setdefault() (dict method)
setdefaultencoding() (in module sys)
setdefaulttimeout() (in module socket)
setdlopenflags() (in module sys)
setDocumentLocator()
    (xml.sax.handler.ContentHandler method)
setDTDHandler() (xml.sax.xmlreader.XMLReader method)
status() (imaplib.IMAP4 method)
statvfs
    module
    statvfs() (module os)
    statvfs() (in module subprocess)
STD_ERROR_HANDLE (in module subprocess)
STD_INPUT_HANDLE (in module subprocess)
STD_OUTPUT_HANDLE (in module subprocess)
StdButtonBox (class in Tix)
stderr (in module sys)
    (subprocess.Popen attribute)
stdin (in module sys)
    (subprocess.Popen attribute)
stdio
STDOUT (in module subprocess)
stdout (in module sys)
    (subprocess.Popen attribute)
Stein, Greg
step (slice object)
    (tkk.Progressbar method)
stereocontrols() (ossaudiodev.oss_mixer_device method)
STILL (in module cd)
stop (slice object)
    (tkk.Progressbar method)
    (unittest.TestResult method)
STOP_CODE (opcode)
stop_here() (bdb.Bdb method)
StopIteration
    exception, [1]
stopListening() (unittest.TestResult method)
stopTest() (unittest.TestResult method)
stopTestRun() (unittest.TestResult method)
storbinary() (ftp
setegid() (in module os)
setEncoding() (xml.sax.xmlreader.InputSource method)
setEntityResolver() (xml.sax.xmlreader.XMLReader method)
setErrorHandler() (xml.sax.xmlreader.XMLReader method)
seteuid() (in module os)
setFeature() (xml.sax.xmlreader.XMLReader method)
setfirstweekday() (in module calendar)
setfmt() (ossaudiodev.oss_audio_device method)
setFormatter() (logging.Handler method)
setframerate() (aifc.aifc method)
   (sunau.AU_write method)
   (wave.Wave_write method)
setgid() (in module os)
setgroups() (in module os)
seth() (in module turtle)
setheading() (in module turtle)
SetInteger() (msilib.Record method)
setItem() (in module operator)
setitimer() (in module signal)
setlast() (mhlb.Folder method)
setLevel() (logging.Handler method)
   (logging.Logger method)
setliteral() (sgmllib.SGMLParser method)
setlocale() (in module locale)
setLocale() (xml.sax.xmlreader.XMLReader method)
setLoggerClass() (in module logging)
setlogmask() (in module syslog)
setmark() (aifc.aifc method)
setMaxConns() (urllib2.CacheFTPHandler method)
setmode() (in module msvcrt)
setName() (threading.Thread method)
setnchannels() (aifc.aifc method)
   (sunau.AU_write method)
   (wave.Wave_write method)
setnframes() (aifc.aifc method)
store() (imaplib.
STORE_ACTIONS (optparse.Option attribute)
STORE_ATTR (opcode)
STORE_DEREF (opcode)
STORE_FAST (opcode)
STORE_GLOBAL (opcode)
STORE_MAP (opcode)
STORE_NAME (opcode)
STORE_SLICE (opcode)
STORE_SLICE (opcode)
STORE_SLICE (opcode)
STORE_SLICE (opcode)
STORE_SUBS (opcode)
storlines() (ftplib
str
   built-in funct
   format
str (built-in clas:
str() (in module
strcoll() (in mod
StreamError
StreamHandler
StreamReader
StreamReader
StreamReader
StreamRecoder
StreamRequestHandler SocketServer
streams
   stackable
StreamWriter (c
strerror() (in module os
strftime() (datetime.date
   (datetime.datetime
   (datetime.time
   (in module
time
strict (csv.Dialect
strict_domain (c
(sunau.AU_write method)
(wave.Wave_write method)
setnomoretags() (sgmlib.SGMLParser method)
setoption() (in module jpeg)
SetParamEntityParsing() (xml.parsers.expat.xmlparser method)
setparameters() (ossaudiodev.oss_audio_device method)
setparams() (aifc.aifc method)
  (in module al)
  (sunau.AU_write method)
  (wave.Wave_write method)
setpassword() (zipfile.ZipFile method)
setpath() (in module fm)
setpgid() (in module os)
setpgrp() (in module os)
setpos() (aifc.aifc method)
  (in module turtle)
  (sunau.AU_read method)
  (wave.Wave_read method)
setPosition() (in module turtle)
setprofile() (in module sys)
  (in module threading)
SetProperty() (msilib.SummaryInformation method)
setProperty() (xml.sax.xmlreader.XMLReader method)
setPublicId() (xml.sax.xmlreader.InputSource method)
setquota() (imaplib.IMAP4 method)
setraw() (in module tty)
setrecursionlimit() (in module sys)
setregid() (in module os)
setresgid() (in module os)
setresuid() (in module os)
setrlimit() (in module resource)
sets (module)
setsampwidth() (aifc.aifc method)
  (sunau.AU_write method)

attribute)
strict_errors() (in
strict_ns_domain
(cookielib.DefaultCookiePolicy attribute)
strict_ns_set_initial_dollar
(cookielib.DefaultCookiePolicy attribute)
strict_ns_unverifiable
(cookielib.DefaultCookiePolicy attribute)
strict_rfc2965_unverifiable
(cookielib.DefaultCookiePolicy attribute)
strides (memory)

string

  Unicode
  comparison
  conversion,
  formatting
  interpolation
  item
  methods
  module, [1],
  object, [1], [:
STRING (in module
string (module)
  (re.MatchObject attribute)
string literal
string_at() (in module
StringIO (class
  (class in StringIO)
(module)
StringIO() (in module
type, strings, document
StringTypes (in module
strip() (in module
setwatchcursor() (in module FrameWork)
setworldcoordinates() (in module turtle)
setx() (in module turtle)
sety() (in module turtle)
SF_APPEND (in module stat)
SF_ARCHIVED (in module stat)
SF_IMMUTABLE (in module stat)
SF_NOUNLINK (in module stat)
SF_SNAPSHOT (in module stat)
SGML

gmgllib
    module
gmgllib (module)
SGMLParseError
SGMLParser (class in gmgllib)
    (in module gmgllib)
sha (module)
Shape (class in turtle)
shape (memoryview attribute)
shape() (in module turtle)
shapesize() (in module turtle)
shared_object_filename()
    (distutils.ccompiler.CCompiler method)
Shelf (class in shelf)

shelve
    module
shelve (module)
shift() (decimal.Context method)
    (decimal.Decimal method)
shift_path_info() (in module wsgiref.util)

shifting
    operation
    operations
shlex (class in shlex)
    (module)
shortDescription() (unittest.TestCase method)
shouldFlush() (logging.handlers.BufferingHandler method)
suppression
(logging.handlers.MemoryHandler method)
shouldStop (unittest.TestResult attribute)
show() (curses.panel.Panel method)
show_choice() (in module fl)
show_compilers() (in module distutils.ccompiler)
show_file_selector() (in module fl)
show_form() (fl.form method)
show_input() (in module fl)
show_message() (in module fl)
show_question() (in module fl)

newline
swapcase() (in str method)
sym() (dl.dl method)
sym_name (in module symbol)
Symbol (class in symbol)
symbol (module)
SymbolTable (class in symtable)
symlink() (in module os)
symmetric_difference() (set method)
symmetric_difference_update() (set method)
symtable (module)
symtable() (in module symtable)
sync() (bsddb.bdbhash)
(dumbdbm.dumbdbm)
(ossaudiodevice)
(shelve.Shelf)
synccdown() (curses.window method)
synchronized() (in module multiprocessing)
SyncManager (multiprocessing)
syncok() (curses.window method)
syncup() (curses.window method)
syntax, [1]
SyntaxErr
SyntaxError
SyntaxWarning
sys
    module, [1],
sys (module)
sys.exc_info
sys.exc_traceback
sys.last_traceback
sys.meta_path
sys.modules
sys.path
sys.path_hooks
sys.path_importer_cache
sys.stderr
sys.stdin
sys.stdout
sys_exec (2to3 fixer)
sys_version
(BaseHTTPServer.BaseHTTPRequestHandler attribute)
SysBeep() (in module MacOS)
sysconf() (in module os)
sysconf_names
sysconfig (module)
syslog (module)
syslog() (in module syslog)
SysLogHandler (class in logging.handlers)
system() (in module os)
(system_alias() (in module platform)
SystemError (built-in exception)
SystemExit (built-in exception)
systemId (xml.dom.Document attribute)
SystemRandom (class in random)
SystemRoot
Index – T

T_FMT (in module locale)
T_FMT_AMPM (in module locale)
tab
tab() (ttk.Notebook method)
TabError
tabnanny (module)
tabs() (ttk.Notebook method)
tabular
data
tag (xml.etree.ElementTree.Element attribute)
tag_bind() (ttk.Treeview method)
tag_configure() (ttk.Treeview method)
tag_has() (ttk.Treeview method)
tagName (xml.dom.Element attribute)
tail (xml.etree.ElementTree.Element attribute)
takewhile() (in module itertools)
TalkTo (class in aetools)
tan() (in module cmath)
(in module math)
tanh() (in module cmath)
(in module math)
TarError
TarFile (class in tarfile), [1]
tarfile (module)
TarFileCompat (class in tarfile)
TarFileCompat.TAR_GZIPPED (in module tarfile)
TarFileCompat.TAR_PLAIN (in module tarfile)
target
deletion
list, [1]
list assignment
tix_cget() (Tix.tixCommand method)
tix_configure()
(Tix.tixCommand method)
tix_filename()
(Tix.tixCommand method)
tix_getbitmap()
(Tix.tixCommand method)
tix_getimage()
(Tix.tixCommand method)
TIX_LIBRARY

tix_option_get()
(Tix.tixCommand method)
tix_resetoptions()
(Tix.tixCommand method)
tixCommand (class in Tix)
Tk

(Tclass in Tkinter)
Tk Option Data Types
TK_LIBRARY
Tkinter

(module)
TList (class in Tix)
TLS
TMP, [1]
TMP_MAX (in module os)
TMPDIR, [1]
tmpfile() (in module os)
tmpnam() (in module os)
to_eng_string()
(decimal.Context method)
(to_integral_exact()
(decimal.Decimal method)
to_integral() (decimal.Decimal method)


list, deletion
loop control
target (xml.dom.ProcessingInstruction attribute)
TarInfo (class in tarfile)
task_done()
(multiprocessing.JoinableQueue method)
(TaskQueue.Queue method)
tb_frame (traceback attribute)
tb_lasti (traceback attribute)
tb_lineno (traceback attribute)
tb_lineno() (in module traceback)
tb_next (traceback attribute)
tcdrain() (in module termios)
tcflow() (in module termios)
tcflush() (in module termios)
tcgetattr() (in module termios)
tcgetpgrp() (in module os)
Tcl() (in module Tkinter)
TCL_LIBRARY
TCPServer (class in SocketServer)
tcsendbreak() (in module termios)
tcgetattr() (in module termios)
tcgetpgrp() (in module os)
tearDown() (unittest.TestCase method)
tearDownClass() (unittest.TestCase method)
(tee() (in module itertools)
tell() (aifc.aifc method), [1]
   (bz2.BZ2File method)
   (chunk.Chunk method)
   (file method)
   (io.IOBase method)
   (io.TextIOBase method)
   (mmap.mmap method)
   (multifile.MultiFile method)
   (sunau.AU_read method)
   (decimal.Context method)
   (decimal.Decimal method)
to_integral_value()
   (decimal.Decimal method)
to_sci_string()
   (decimal.Context method)
to_splittable()
   (email.charset.Charset method)
   ToASCII() (in module encodings.idna)
tobuf() (tarfile.TarInfo method)
tobytes() (memoryview method)
tochild (popen2.Popen3 attribute)
today() (datetime.date class method)
   (datetime.datetime method)
tofile() (array.array method)
tok_name (in module token)
token
   (module)
   (shlex.shlex attribute)
tokeneater() (in module tabnanny)
TokenError
tokenize (module)
tokenize() (in module token)
tolist() (array.array method)
   (memoryview method)
   (parser.ST method)
tomono() (in module audio)
toordinal() (datetime.date method)
   (datetime.datetime method)
Telnet (class in telnetlib)
telnetlib (module)

TEMP

tempdir (in module tempfile)
tempfile (module)

Template (class in pipes)
     (class in string)
template (in module tempfile)
     (string.Template attribute)

tempnam() (in module os)
temporary
     file
     file name

TemporaryFile() (in module tempfile)

TERM, [1]
termattrs() (in module curses)
terminate()

(multiprocessing.pool.multiprocessing.Pool method)
     (multiprocessing.Process method)
     (subprocess.Popen method)
termination model
termios (module)
termmame() (in module curses)
ternary
     operator
test
     identity
     membership
test (doctest.DocTestFailure attribute)
     (doctest.UnexpectedException attribute)
     (module)
test() (in module cgi)

top() (curses.panel.Panel method)
     (poplib.POP3 method)
top_panel() (in module curses.panel)
toprettyxml()
     (xml.dom.minidom.Node method)
tostereo() (in module audioop)
tostring() (array.array method)
     (in module xml.etree.ElementTree)
tostringlist() (in module xml.etree.ElementTree)
total_changes
     (sqlite3.Connection attribute)
total_ordering() (in module functools)
total_seconds()
     (datetime.timedelta method)
totuple() (parser.ST method)
touched() (in module macostools)
touchline() (curses.window method)
touchwin() (curses.window method)
tounicode() (array.array method)
     ToUnicode() (in module encodings.idna)
tovideo() (in module imageop)
towards() (in module turtle)
toxml()
     (xml.dom.minidom.Node method)
     tp_as_mapping (C member)
mutex.mutex method

test.test_support (module)
testandset() (mutex.mutex method)
TestCase (class in unittest)
TestFailed
testfile() (in module doctest)
TESTFN (in module test.test_support)
TestLoader (class in unittest)
testMethodPrefix (unittest.TestLoader attribute)
testmod() (in module doctest)
TestResult (class in unittest)
tests (in module imghdr)
testsRun (unittest.TestResult attribute)
TestSuite (class in unittest)
testzip() (zipfile.ZipFile method)
text (in module msilib)
   (xml.etree.ElementTree.Element attribute)
text() (msilib.Dialog method)
text_factory (sqlite3.Connection attribute)
Textbox (class in curses.textpad)
TextCalendar (class in calendar)
textdomain() (in module gettext)
TextFile (class in distutils.text_file)
TextIOBase (class in io)
TextIOWrapper (class in io)
TextTestResult (class in unittest)
TextTestRunner (class in unittest)
textwrap (module)
TextWrapper (class in textwrap)
   theme_create() (ttk.Style method)
   theme_names() (ttk.Style method)
   theme_settings() (ttk.Style method)
   theme_use() (ttk.Style method)

THOUSEP (in module locale)

thread

tp_as_number (C member)

tp_as_sequence (C member)
tparm() (in module curses)

trace
   stack
Trace (class in trace)
trace (module)

trace command line option
   --help
   --ignore-dir=<dir>
   --ignore-module=<mod>
   --version
   -C, --coverdir=<dir>
   -R, --no-report
   -T, --trackcalls
   -c, --count
   -f, --file=<file>
   -g, --timing
   -I, --listfuncs
   -m, --missing
   -r, --report
   -s, --summary
   -t, --trace

trace function, [1], [2]
trace() (in module inspect)
trace_dispatch() (bdb.Bdb method)

traceback
   object, [1], [2], [3], [4]
traceback (module)
traceback_limit
   (wsgiref.handlers.BaseHandler attribute)
tracebacklimit (in module sys)

tracebacks
   in CGI scripts
module
Thread (class in threading)
thread (module)
thread() (imaplib.IMAP4 method)
ThreadError
threading (module)
ThreadingMixIn (class in SocketServer)
ThreadingTCPServer (class in SocketServer)
ThreadingUDPServer (class in SocketServer)
threads
   IRIX
   POSIX
throw (2to3 fixer)
throw() (generator method)
tie() (in module fl)
tigetflag() (in module curses)
tigetnum() (in module curses)
tigetstr() (in module curses)
TILDE (in module token)
tilt() (in module turtle)
tiltangle() (in module turtle)
time (class in datetime)
   (module)
time() (datetime.datetime method)
   (in module time)
Time2Internaldate() (in module imaplib)
timedelta (class in datetime)
TimedRotatingFileHandler (class in logging.handlers)
timegm() (in module calendar)
timeit (module)
timeit command line option
   -c, --clock
   -h, --help
   -n N, --number=N
TracebackType (in module types)
tracer() (in module turtle), [1
trailing
   comma, [1]
transfercmd() (ftplib.FTP method)
TransientResource (class in test.test_support)
translate() (in module fnmatch)
   (in module string)
   (str method)
translation() (in module gettext)
Transport Layer Security
traverseproc (C type)
Tree (class in Tix)
TreeBuilder (class in xml.etree.ElementTree)
Treeview (class in ttk)
triangular() (in module random)
triple-quoted string, [1
True, [1], [2]
true
True (built-in variable)
truediv() (in module operator)
trunc() (in module math), [1
truncate() (file method)
   (io.IOBase method)
truth
   value
truth() (in module operator)
try
   statement, [1], [2]
ttk
   (module)	tob() (in module imgfile)
-r N, --repeat=N
-s S, --setup=S
-t, --time
-v, --verbose

timeit() (in module timeit)
(timeit.Timer method)

timeout
(SocketServer.BaseServer attribute)
timeout() (curses.window method)

Timer (class in threading)

times() (in module os)
timetuple() (datetime.date method)

(timeit.Timer method)
timetuple() (datetime.datetime method)
timezone (in module time)
title() (EasyDialogs.ProgressBar method)

(tuple_params (2to3 fixer)
TupleType (in module types)

turnoff_sigfpe() (in module fpectl)
turnon_sigfpe() (in module fpectl)

Turtle (class in turtle)
turtle (module)
turtles() (in module turtle)
TurtleScreen (class in turtle;
turtlesize() (in module turtle;

Tutt, Bill
type, [1]

Boolean

built-in function, [1], [2], [3], [4]
data hierarchy
immutable data
object, [1], [2]
operations on dictionary
operations on list
type (built-in class)
Type (class in aetypes)
type (optparse.Option

tty

I/O control
tty (module)
ttymame() (in module os)
tuple

built-in function, [1]
display
empty, [1]
object, [1], [2], [3], [4], [5]
singleton
tuple() (built-in function)
tuple2st() (in module parser)
tuple_params (2to3 fixer)

Turtle (class in turtle)
turtle (module)
turtles() (in module turtle)
TurtleScreen (class in turtle;
turtlesize() (in module turtle;

Tutt, Bill
type, [1]
attribute)
    (socket.socket attribute)
    (tarfile.TarInfo attribute)
type of an object
TYPE_CHECKER
(optparse.Option attribute)
typeahead() (in module curses)
typecode (array.array attribute)
TYPED_ACTIONS
(optparse.Option attribute)
typed_subpart_iterator() (in module email.iterators)
TypeError
    exception
types
    built-in
    module
    mutable sequence
    operations on integer
    operations on mapping
    operations on numeric
    operations on sequence
[1]
types (2to3 fixer)
    (module)
TYPES (optparse.Option attribute)
types, internal
types_map (in module mimetypes)
    (mimetypes.MimeTypes attribute)
types_map_inv
    (mimetypes.MimeTypes attribute)
TypeError (in module types)
TZ, [1], [2], [3], [4]
tzinfo (class in datetime)
  (datetime.datetime attribute)
  (datetime.time attribute)
tzname (in module time)
tzname() (datetime.datetime method)
  (datetime.time method)
  (datetime.tzinfo method)
tzset() (in module time)

[1]
Index – U

U (in module re)
u-LAW, [1], [2], [3]
ucd_3_2_0 (in module unicodedata)
udata (select.kevent attribute)
UDPServer (class in SocketServer)
UF_APPEND (in module stat)
UF_COMPRESSED (in module stat)
UF_HIDDEN (in module stat)
UF_IMMUTABLE (in module stat)
UF_NODUMP (in module stat)
UF_NOUNLINK (in module stat)
UF_OPAQUE (in module stat)
uggettext() (gettext.GNUTranslations method)
           (gettext.NullTranslations method)
uid (tarfile.TarInfo attribute)
uid() (imaplib.IMAP4 method)
uidl() (poplib.POP3 method)
ulaw2lin() (in module audioop)
ULONG_MAX
umask() (in module os)
uname (tarfile.TarInfo attribute)
uname() (in module os)
       (in module platform)

unary
   arithmetic operation
   bitwise operation
UNARY_CONVERT (opcode)
UNARY_INVERT (opcode)
UNARY_NEGATIVE (opcode)

UNPACK_SEQUENCE (opcode)
unpack_string() (xdrlib.Unpacker method)
Unpacker (class in xdrlib)
unpackevent() (in module aertools)
unparsedEntityDecl() (xml.sax.handler.DTDHandler method)
UnparsedEntityDeclHandler() (xml.parsers.expat.xmlparser method)
Unpickler (class in pickle)
UnpicklingError
unqdevice() (in module fl)
unquote() (in module email.utils)
       (in module rfc822)
       (in module urllib)
unquote_plus() (in module urllib)
unreachable object
unreadline() (distutils.text_file.TextFile method)
unrecognized escape sequence
unregister() (select.epoll method)
       (select.poll method)
unregister_archive_format() (in module shutil)
unregister_dialect() (in module csv)
unset()
       (test.test_support.EnvironmentVarGuard method)
unsetenv() (in module os)
unsubscribe() (imaplib.IMAP4 method)
UnsupportedOperation
untokenize() (in module tokenize)
untouchwin() (curses.window method)
unused_data (zlib.Decompress attribute)
UNARY_NOT (opcode)
UNARY_POSITIVE (opcode)
unbinding

UnboundLocalError, [1]
UnboundMethodType (in module
types)
unbuffered I/O
UNC paths
    and os.makedirs()
unconsumed_tail
(zlib.Decompress attribute)
unctrl() (in module curses)
    (in module curses.ascii)
undefine_macro()
(distutils.ccompiler.CCompiler
method)
Underflow (class in decimal)
undo() (in module turtle)
undobufferentries() (in module
turtle)
undoc_header (cmd.Cmd
attribute)
unescape() (in module
xml.sax.saxutils)
UnexpectedException
unexpectedSuccesses
(unitest.TestResult attribute)
unfreeze_form() (fl.form method)
ungetch() (in module curses)
    (in module msvcrt)
ungetmouse() (in module curses)
ungettext()
 gettext.GNUTranslations
method)
    (gettext.NullTranslations
method)
ungetwch() (in module msvcrt)
unwrap() (ssl.SSLSocket method)
up() (in module turtle)
update() (collections.Counter
method)
    (dict method)
    (hashlib.hash method)
    (hmac.HMAC method)
    (in module turtle)
    (mailbox.Mailbox method)
    (mailbox.Maildir method)
    (md5.md5 method)
    (set method)
    (sha.sha method)
    (trace.CoverageResults method)
update_panels() (in module
curses.panel)
update_visible()
    (mailbox.BabylMessage method)
update_wrapper() (in module
functools)
updatescrollbars()
    (FrameWork.ScrolledWindow
method)
upper() (in module string)
    (str method)
upercase (in module string)
urandom() (in module os)
URL, [1], [2], [3], [4]
    parsing
    relative
url (xmlrpclib.ProtocolError
attribute)
url2pathname() (in module urllib)
urllcleanup() (in module urllib)
urldefrag() (in module urllib)
urlencode() (in module urllib)
URLLError
urljoin() (in module urlparse)
urllib
    module
urllib (2to3 fixer)
unhexlify() (in module binascii)

unichr
  built-in function
unichr() (built-in function)
Unicode, [1], [2]
  database
  object
unicode
  built-in function, [1], [2], [3]
  object
unicode (2to3 fixer)
UNICODE (in module re)
Unicode Consortium
unicode() (built-in function)
unicodedata (module)
UnicodeDecodeError
UnicodeEncodeError
UnicodeError
UnicodeTranslateError
UnicodeType (in module types)
UnicodeWarning
unidata_version (in module unicodedata)
unified_diff() (in module difflib)
uniform() (in module random)
UnimplementedFileMode
uninstall() (imputil.ImportManager method)
Union (class in ctypes)
union() (set method)
unittest (module)
unittest command line option
  -b, --buffer
  -c, --catch
  -f, --failfast
unittest-discover command line option

(user-defined)

urllib2 (module)
urlopen() (in module urllib)
  (in module urllib2)
URLopener (class in urllib)
urllparse
  module
urllparse (module)
urllparse() (in module urlparse)
urlretrieve() (in module urllib)
ursafe_b64decode() (in module base64)
ursafe_b64encode() (in module base64)
urlsplit() (in module urlparse)
urlunparse() (in module urlparse)
urlunsplit() (in module urlparse)
urn (uuid.UUID attribute)
use_default_colors() (in module curses)
use_env() (in module curses)
use_rawinput (cmd.Cmd attribute)
UseForeignDTD()
(xml.parsers.expat.xmlparser method)
USER

user
  configuration file
  effective id
    id
    id, setting
user (module)
user() (poplib.POP3 method)

(user-defined)

function
  function call
  method

user-defined function
  object, [1], [2]
universal newlines
What's new, [1], [2], [3]
bz2.BZ2File class
file.newlines attribute
io.IncrementalNewlineDecoder class
io.TextIOWrapper class
open() (in module io)
open() built-in function
str.splitlines method
subprocess module
zipfile.ZipFile.open method
UNIX
I/O control
file control
UnixDatagramServer (class in SocketServer)
unixfrom (rfc822.Message attribute)
UnixMailbox (class in mailbox)
UnixStreamServer (class in SocketServer)
Unknown (class in aetypes)
unknown_charref()
(sgmllib.SGMLParser method)
unknown_decl()
(HTMLParser.HTMLParser method)
unknown_endtag()
(sgmllib.SGMLParser method)
unknown_entityref()
(struct.Struct method)
unpack_fstring() (xdrlib.Unpacker method)
unpack_list() (xdrlib.Unpacker method)
unpack_opaque()
(xdrlib.Unpacker method)
validator() (in module wsgiref.validate)

value
default parameter
truth
value (Cookie.Morsel attribute)
(cookielib.Cookie attribute)
(ctypes._SimpleCData attribute)
(xml.dom.Attr attribute)
value of an object
Value() (in module multiprocessing)
(in module multiprocessing.sharedctypes)
(multiprocessing.managers.SyncManager method)
value_decode() (Cookie.BaseCookie method)
value_encode() (Cookie.BaseCookie method)
ValueError exception
valuerefs() (weakref.WeakValueDictionary method)
values
Boolean
writing, [1]
values() (dict method)
(email.message.Message method)
(mailbox.Mailbox method)
ValuesView (class in collections)
variable
free, [1]
variant (uuid.UUID attribute)
varray() (in module gl)
vars() (built-in function)
VBAR (in module token)

VERIFY_DEFAULT (in module ssl)
verify_flags (ssl.SSLContext attribute)
verify_mode (ssl.SSLContext attribute)
verify_request() (SocketServer.BaseServer method)
VERIFY_X509 STRICT (ssl)
VERIFY_X509_TRUSTED_FIRST (ssl)
version (cookielib.Cookie)
(httplib.HTTPResponse method)
(in module curses)
(in module marshal)
(in module sqlite3)
(in module sys), [1], [2]
(urllib.URLopener attribute)
(uuid.UUID attribute)
version() (in module ensurepip)
(in module platform)
(ssl.SSLSocket method)
version_info (in module sqlite3)
(in module sys)
version_string() (BaseHTTPServer.BaseHTTPServer method)
vformat() (string.Formatter method)
videoreader (module)
viewitems() (dict method)
viewkeys() (dict method)
viewvalues() (dict method)
virtual environment
virtual machine
visit() (ast.NodeVisitor method)
visitproc (C type)
vline() (curses.window method)
vbar (ScrolledText.ScrolledText attribute)
VBAREQUAL (in module token)
Vec2D (class in turtle)
VERBOSE (in module re)
verbose (in module tabnanny)
  (in module test.test_support)
verify() (smtplib.SMTP method)
VERIFY_CRL_CHECK_CHAIN (in module ssl)
VERIFY_CRL_CHECK_LEAF (in module ssl)

VMSError
vnarray() (in module gi)
voidcmd() (ftplib.FTP method)
volume (zipfile.ZipInfo attribute)
vonmisesvariate() (in module...
Index – W

W (module)
W_OK (in module os)
wait() (in module os)
  (multiprocessing.pool.AsyncResult method)
  (popen2.Popen3 method)
  (subprocess.Popen method)
  (threading.Condition method)
  (threading.Event method)
wait3() (in module os)
wait4() (in module os)
waitpid() (in module os)
walk() (email.message.Message method)
  (in module ast)
  (in module compiler)
  (in module compiler.visitor)
  (in module os)
  (in module os.path)
walk_packages() (in module pkgutil)
want (doctest.Example attribute)
warn() (distutils.ccompiler.CCompiler method)
  (distutils.text_file.TextFile method)
  (in module warnings)
warn_explicit() (in module warnings)
Warning
warning() (in module logging)
  (logging.Logger method)
  (xml.sax.handler.ErrorHandler method)
warnings
  (module)
WarningsRecorder (class in test.test_support)
warnoptions (in module sys)
warnpy3k() (in module warnings)
wasSuccessful() (unittest.TestResult method)

WinSock
winsound (module)
winver (in module sys)

with
  statement, [1]
  WITH_CLEANUP (openg
  WMAvailable() (in module
  WNOHANG (in module os)
  wordchars (shlex.shlex as
  World Wide Web, [1], [2]
  wrap() (in module textwr
  (textwrap.TextWrap
  wrap_socket() (in module
  (ssl.SSLContext method)
  wrap_text() (in module
distutils.fancy_getopt)
  wrapper() (in module cu
  wraps() (in module funct
  WRITABLE (in module T
  writable() (asyncore.dis
  (io.IOBase method)
  write() (array.array meth
  (ConfigParser.RawConfig
  (bz2.BZ2File method
code.InteractiveInterp
codecs.StreamWrite
codecs.StreamReader
(email.generator.Gem
(file method)
  (in module imgfile)
  (in module os)
  (in module turtle)
  (io.BufferedIOBase met
WatchedFileHandler (class in logging.handlers)
wave (module)
WCONTINUED (in module os)
WCOREDUMP() (in module os)
WeakKeyDictionary (class in weakref)
weakref (module)
WeakSet (class in weakref)
WeakValueDictionary (class in weakref)
webbrowser (module)
weekday() (datetime.date method)
    (datetime.datetime method)
    (in module calendar)
weekheader() (in module calendar)
weibullvariate() (in module random)
WEXITSTATUS() (in module os)
while (BaseHTTPServer.BaseHTTPRequestHandler attribute)
what() (in module imghdr)
    (in module sndhdr)
whathdr() (in module sndhdr)
whichdb (module)
whichdb() (in module whichdb)
while
    statement, [1], [2], [3]
whitespace
    (in module string)
    (shlex.shlex attribute)
whitespace_split (shlex.shlex attribute)
whseed() (in module random)
WichmannHill (class in random)
Widget (class in ttk)
width (textwrap.TextWrapper attribute)
width() (in module turtle)
WIFCONTINUED() (in module os)
WIFEXITED() (in module os)
WIFSIGNALED() (in module os)
write_byte() (mmap.mmap method)
write_docstringdict() (in )
write_file() (in module distutils.file_util)
write_history_file() (in module readline)
WRITE_RESTRICTED
write_results() (trace.CoverageResults method)
writeall() (ossaudiodev.oss_audio_device method)
writebufferproc (C type)
writeframes() (aiffc.aiffc method)
    (sunau.AU_write method)
    (wave.Wave_write method)
writeframesraw() (aiffc.aiffc method)
    (sunau.AU_write method)
    (wave.Wave_write method)
writeheader() (csv.DictWriter method)
writelines() (bz2.BZ2File method)
    (codecs.StreamWriter method)
    (file method)
writePlist() (in module plistlib)
writePlistToResource() (in module plistlib)
writePlistToString() (in module plistlib)
writepy() (zipfile.PyZipFile method)
writer (formatter.formatter attribute)
WIFSTOPPED() (in module os)
Wimp$ScrapDir
win32_ver() (in module platform)
WinDLL (class in ctypes)
window manager (widgets)
window() (curses.panel.Panel method)
Window() (in module FrameWork)
window_height() (in module turtle), [1]
window_width() (in module turtle), [1]
windowbounds() (in module FrameWork)
Windows ini file
WindowsError
WinError() (in module ctypes)
WINFUNCTYPE() (in module ctypes)

writer() (in module csv)
writerow() (csv.csvwriter method)
writerows() (csv.csvwriter method)
writestr() (zipfile.ZipFile method)
writexml() (xml.dom.minidom method)
writing
WrongDocumentErr
ws_comma (2to3 fixer)
wsgi_file_wrapper
(wsgiref.handlers.BaseHandler attribute)
wsgi_multiprocess
(wsgiref.handlers.BaseHandler attribute)
wsgi_multithread
(wsgiref.handlers.BaseHandler attribute)
wsgi_run_once
(wsgiref.handlers.BaseHandler attribute)
wsgiref (module)
wsgiref.handlers (module)
wsgiref.headers (module)
wsgiref.simple_server (module)
wsgiref.util (module)
wsgiref.validate (module)
WSGIRequestHandler (wsgiref.simple_server)
WSGIServer (class in wsgiref.simple_server)
wShowWindow
(subprocess.STARTUPINFO attribute)
WSTOPSIG() (in module os)
wstring_at() (in module ctypes)
WTERMSIG() (in module os)
WUNTRACED (in module os)
WWW, [1], [2], [3]

server, [1]
## Index – X

| X (in module re)                     | XML_NAMESPACE (in module xml.dom)                        |
| X509 certificate                     | xmlcharrefreplace_errors() (in module codecs)            |
| X_OK (in module os)                  | XmlDeclHandler() (xml.parsers.expat.xmlparser method)    |
| xatom() (imaplib.IMAP4 method)       | XMLFilterBase (class in xml.sax.saxutils)               |
| xcor() (in module turtle)            | XMLGenerator (class in xml.sax.saxutils)                |
| XDR, [1]                             | XMLID() (in module xml.etree.ElementTree)               |
| xdrlib (module)                      | XMLNS_NAMESPACE (in module xml.dom)                     |
| xgtitle() (nntplib.NNTP method)      | XMLParser (class in xml.etree.ElementTree)              |
| xhdr() (nntplib.NNTP method)         | XMLParserType (in module xml.parsers.expat)             |
| XHTML                                 | XMLReader (class in xml.sax.xmlreader)                  |
| XHTML_NAMESPACE (in module xml.dom)  | xmrpclib (module)                                       |
| xml (module)                          | xor [1]                                                |
| XML() (in module xml.etree.ElementTree) | bitwise     |
| xml.dom (module)                      | xor() (in module operator)                              |
| xml.dom.minidom (module)             | xover() (nntplib.NNTP method)                            |
| xml.dom.pulldom (module)             | xpath() (nntplib.NNTP method)                           |
| xml.etree.ElementTree                | xrange [1]                                             |
| (module)                             | built-in function                                       |
| xml.parsers.expat (module)           | object, [1]                                            |
| xml.sax (module)                     | xrange (2to3 fixer)                                     |
| xml.sax.handler (module)             | xrange() (built-in function)                            |
| xml.sax.saxutils (module)            | XRangeType (in module types)                            |
| xml.sax.xmlreader (module)           | xreadlines (2to3 fixer)                                  |
|                                       | xreadlines() (bz2.BZ2File method)                        |
(file method)
xview() (ttk.Treeview method)
## Index – Y

<table>
<thead>
<tr>
<th>Y2K</th>
<th>yeardayscalendar()</th>
</tr>
</thead>
<tbody>
<tr>
<td>ycor() (in module turtle)</td>
<td>(calendar.Calendar method)</td>
</tr>
<tr>
<td>year (datetime.date attribute)</td>
<td>YESEXPR (in module locale)</td>
</tr>
<tr>
<td>(datetime.datetime attribute)</td>
<td>yield expression</td>
</tr>
<tr>
<td>Year 2000</td>
<td>keyword statement</td>
</tr>
<tr>
<td>Year 2038</td>
<td>YIELD_VALUE (opcode)</td>
</tr>
<tr>
<td>yeardatescalendar()</td>
<td>yiq_to_rgb() (in module colorsys)</td>
</tr>
<tr>
<td>(calendar.Calendar method)</td>
<td>yview() (ttk.Treeview method)</td>
</tr>
</tbody>
</table>
Index – Z

Zen of Python
ZeroDivisionError
exception
zfill() (in module string)
    (str method)
zip (2to3 fixer)
zip() (built-in function)
    (in module future_builtins)
ZIP_DEFLATED (in module zipfile)
ZIP_STORED (in module zipfile)
ZipFile (class in zipfile)

zipfile (module)
zipfile command line option
    -c  <zipfile>  <source1>  ...  <sourceN>
    -e  <zipfile>  <output_dir>
    -l  <zipfile>
    -t  <zipfile>
zipimport (module)
zipimporter (class in zipimport)
ZipImportError
ZipInfo (class in zipfile)
zlib (module)
Index

Symbols | _ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
Symbols

!=
   operator
%
   operator
% formatting
% interpolation
%=
   augmented assignment
%PATH%
&
   operator
&=
   augmented assignment
*
   in function calls
operator
statement, [1]
**
   in function calls
operator
statement, [1]
**=
   augmented assignment
*= 
   augmented assignment
+
   operator
+=
   augmented assignment
-
   operator
--help
command line option
-p, --pattern pattern
unittest-discover command line option
-q
compileall command line option
-Q <arg>
   command line option
-R
command line option
-r N, --repeat=N
timeit command line option
-R, --no-report
trace command line option
-r, --report
trace command line option
-S
command line option
-s
command line option
-s S, --setup=S
timeit command line option
-s, --start-directory directory
unittest-discover command line option
-s, --summary
trace command line option
-t
command line option
-t <zipfile>
zipfile command line option
-t, --time
timeit command line option
trace command line option
--ignore-dir=<dir>
trace command line option
--ignore-module=<mod>
trace command line option
--user-base
site command line option
--user-site
site command line option
--version
command line option
-3
command line option
-=
augmented assignment
-?
command line option
-B
command line option
-b, --buffer
unittest command line option
-c <command>
command line option
-c <zipfile> <source1> ...
<sourceN>
zipfile command line option
-c, --catch
unittest command line option
-c, --clock
timeit command line option
-c, --count
trace command line option
-C, --coverdir=<dir>
trace command line option
-d
-t, --top-level-directory directory
unittest-discover command line option
-t, --trace
trace command line option
-T, --trackcalls
trace command line option
-U
command line option
-u
command line option
-V
command line option
-v
command line option
-v, --verbose
timeit command line option
unittest-discover command line option
-W arg
command line option
-X
command line option
-x
command line option
-x regex
compileall command line option
...
command line option
-d destdir
compileall command line option
-E
command line option
-e <zipfile> <output_dir>
zipfile command line option
-f
compileall command line option
-f, --failfast
unittest command line option
-f, --file=<file>
trace command line option
-g, --timing
trace command line option
-h
command line option
-h, --help
timeit command line option
-i
command line option
-i list
compileall command line option
-J
command line option
-l
compileall command line option
-l <zipfile>
zipfile command line option
-l, --listfuncs
trace command line option
-m <module-name>
command line option

-m, --missing
trace command line option

-n N, --number=N
timeit command line option

-O
command line option

-OO
command line option

augmented assignment
|= augmented assignment
__abs__() (in module operator)  
(object method)  
__add__() (in module operator)  
(object method)  
__all__  
(optional module attribute)  
(package variable)  
__and__() (in module operator)  
(object method)  
__bases__ (class attribute),  
[1]  
__builtin__  
(module, [1], [2], [3], [4]  
__builtin__ (module)  
__builtins__  
__call__() (object method),  
[1]  
__class__ (instance attribute), [1]  
__closure__ (function attribute)  
__cmp__() (instance method)  
(object method)  
__code__ (function attribute)  
__coerce__() (object method)  
__complex__() (object method)  
__concat__() (in module operator)  
__oct__() (object method)  
__or__() (in module operator)  
(object method)  
__package__  
__path__, [1]  
__pos__() (in module operator)  
(object method)  
__pow__() (in module operator)  
(object method)  
__radd__() (object method)  
__rand__() (object method)  
__rcmp__() (object method)  
__rdiv__() (object method)  
__rdivmod__() (object method)  
__reduce__() (object method)  
__reduce_ex__() (object method)  
__repeat__() (in module operator)  
__repr__()  
(multiprocessing.managers.BaseProxy method)  
(nterc.netrc method)  
(object method)  
__reversed__() (object method)  
__rfloordiv__() (object method)  
__rlshift__() (object method)  
__rmod__() (object method)  
__rmul__() (object method)  
__ror__() (object method)  
__rpow__() (object method)  
__rrshift__() (object method)  
__rshift__() (in module operator)  
(object method)  
__rsub__() (object method)  
__rtruediv__() (object method)
__future__
(module)
__ge__() (in module operator)
    (instance method)
    (object method)
__get__() (object method)
__getattribute__() (object method)
__getinitargs__() (object method)
__getitem__() (email.message.Message method)
    (in module operator)
    (mailbox.Mailbox method)
    (mapping object method)
    (object method)
__getnewargs__() (object method)
__getslice__() (in module operator)
    (object method)
__getstate__() (object method)
__globals__ (function attribute)
__gt__() (in module operator)
    (instance method)
    (object method)
__hash__() (object method)
__hex__() (object method)
iadd__() (in module operator)
    (object method)
(rfc822.AddressList method)
_name (ctypes.PyDLL attribute)
_objects (ctypes._CData attribute)
_pack_ (ctypes.Structure attribute)
_parse() (gettext.NullTranslations method)
_Pointer (class in ctypes)
__Py_c_diff (C function)
__Py_c_neg (C function)
__Py_c_pow (C function)
__Py_c_prod (C function)
__Py_c_quot (C function)
__Py_c_sum (C function)
__Py_NoneStruct (C variable)
__PyImport_FindExtension (C function)
__PyImport_Fini (C function)
__PyImport_FixupExtension (C function)
__PyImport_Init (C function)
__PyObject_Del (C function)
__PyObject_GC_TRACK (C function)
__PyObject_GC_UNTRACK (C function)
__PyObject_New (C function)
__PyObject_NewVar (C function)
__PyString_Resize (C function)
__PyTuple_Resize (C function)
quit() (FrameWork.Application method)
__replace() (collections.somenamedtuple method)
_setroot()
_start() (aetools.TalkTo method)
structure() (in module email.iterators)
type_ (ctypes._Pointer attribute)
    (ctypes.Array attribute)
_urlopener (in module urllib)
_winreg (module)
_write() (wsgiref.handlers.BaseHandler method)
operator)
   __invert__() (in module operator)
       (object method)
   __ior__() (in module operator)
       (object method)
   __ipow__() (in module operator)
       (object method)
   __irepeat__() (in module operator)
       (object method)
   __irshift__() (in module operator)
       (object method)
   __isub__() (in module operator)
       (object method)
       (rfc822.AddressList method)
   __iter__() (container method)
       (iterator method)
       (mailbox.Mailbox method)
       (object method)
       (unittest.TestSuite method)
   __itruediv__() (in module operator)
       (object method)
   __ixor__() (in module operator)
       (object method)
   __le__() (in module operator)
       (instance method)
   __len__() (email.message.Message
method)
    (mailbox.Mailbox method)
    (mapping object method)
    (object method)
    (rfc822.AddressList method)
__loader__
__long__() (object method)
__lshift__() (in module operator)
    (object method)
__lt__() (in module operator)
    (instance method)
    (object method)
__main__
    module, [1], [2], [3], [4], [5], [6], [7]
__main__ (module)
__members__ (object attribute)
__metaclass__ (built-in variable)
__methods__ (object attribute)
__missing__() (collections.defaultdict method)
    (object method)
__mod__() (in module operator)
    (object method)
__module__ (class attribute)
    (function attribute)
    (method attribute)
__mro__ (class attribute)
__mul__() (in module operator)
(object method)
__name__
(class attribute)
(definition attribute)
(function attribute)
(method attribute)
(module attribute), [1], [2]
__ne__()
(email.charset.Charset method)
  (email.header.Header method)
  (in module operator)
  (instance method)
  (object method)
__neg__() (in module operator)
  (object method)
__new__() (object method)
__nonzero__() (object method), [1]
__not__() (in module operator)
A-LAW, [1]
a-LAW
a2b_base64() (in module binascii)
a2b_hex() (in module binascii)
a2b_hqx() (in module binascii)
a2b_qp() (in module binascii)
a2b_uu() (in module binascii)
abc (module)
ABCMeta (class in abc)
abort()
    (ftplib.FTP method)
    (in module os)
above() (curses.panel.Panel method)
abs
    built-in function, [1]
abs() (built-in function)
    (decimal.Context method)
    (in module operator)
abspath() (in module os.path)
abstract base class
AbstractBasicAuthHandler (class in urllib2)
AbstractDigestAuthHandler (class in urllib2)
AbstractFormatter (class in formatter)
abstractmethod() (in module abc)
abstractproperty() (in module abc)
AbstractWriter (class in formatter)
accept() (asyncore.dispatcher method)
    (multiprocessing.connection.Listener method)
    (socket.socket method)
accept2dyear (in module time)
access() (in module os)
acos() (in module cmath)
    (in module math)
acosh() (in module cmath)
anchor_bgn() (htmllib.
anchor_end() (htmllib.
and
    bitwise
    operator, [1], [2]
and_() (in module operator)
annotate() (in module dircache)
announce() (distutils.ccompiler.CCompiler method)
anonymous
    function
answer_challenge() (multiprocessing.connection method)
any() (built-in function)
anydbm (module)
api_version (in module sys)
apop() (poplib.POP3 method)
APPDATA
append() (array.array method)
    (collections.deque method)
    (email.header.Header method)
    (imaplib.IMAP4 method)
    (list method)
    (msilib.CAB method)
    (pipes.Template method)
    (xml.etree.ElementTree.Element method)
appendChild() (xml.dom.Node method)
appendleft() (collections.deque method)
AppleEvents, [1]
applesingle (module)
Application() (in module)
application_uri() (in module)
aply
    built-in function, [1]
add_charset() (in module email.charset)
add_choice() (fl.form method)
add_clock() (fl.form method)
add_codec() (in module email.charset)
add_cookie_header() (cookielib.CookieJar method)
add_counter() (fl.form method)
add_data() (in module msilib)

add_dial() (fl.form method)
add_fallback() (gettext.NullTranslations method)
add_file() (msilib.Directory method)
add_flag() (mailbox.MaildirMessage method)

add_flowing_data() (formatter.formatter method)
add_folder() (mailbox.Maildir method)

add_handler() (urllib2.OpenerDirector method)
add_header() (email.message.Message method)

add_history() (in module readline)
add_hor_rule() (formatter.formatter method)
add_include_dir() (distutils.ccompiler.CCompiler method)
add_input() (fl.form method)
add_label() (mailbox.BabylMessage method)
add_label_data() (formatter.formatter method)
add_library() (distutils.ccompiler.CCompiler method)
add_library_dir() (distutils.ccompiler.CCompiler method)
add_lightbutton() (fl.form method)
add_line_break() (formatter.formatter method)
add_link_object() (distutils.ccompiler.CCompiler method)
add_literal_data() (formatter.formatter method)
add_menu() (fl.form method)
add_mutually_exclusive_group()
(argparse.ArgumentParser method)
add_option() (optparse.OptionParser method)
add_parent() (urllib2.BaseHandler method)
add_password() (urllib2.HTTPPasswordMgr method)
add_positioner() (fl.form method)
add_roundbutton() (fl.form method)
add_runtime_library_dir()
(distutils.ccompiler.CCompiler method)
add_section() (ConfigParser.RawConfigParser method)
add_sequence() (mailbox.MHMessage method)
add_slider() (fl.form method)
add_stream() (in module msilib)
add_subparsers() (argparse.ArgumentParser method)
add_suffix() (imputil.ImportManager method)
add_tables() (in module msilib)
add_text() (fl.form method)
add_timer() (fl.form method)
add_type() (in module mimetypes)
add_unredirected_header() (urllib2.Request method)
add_valslider() (fl.form method)
addch() (curses.window method)
addCleanup() (unittest.TestCase method)
addcomponent() (turtle.Shape method)
addError() (unittest.TestResult method)
addExpectedFailure() (unittest.TestResult method)
addFailure() (unittest.TestResult method)
addfile() (tarfile.TarFile method)
adddFilter() (logging.Handler method)
    (logging.Logger method)
addHandler() (logging.Logger method)
addheader() (MimeWriter.MimeWriter method)
assertFalse() (unittest.TestCase method)
assertGreater() (unittest.TestCase method)
assertGreaterEqual() (unittest.TestCase method)
assertIn() (unittest.TestCase method)
AssertionError
    exception
assertions
debugging
assertEqual() (unittest.TestCase method)
assertException() (unittest.TestCase method)
assertFailure() (unittest.TestCase method)
assertIs() (unittest.TestCase method)
assertIsInstance() (unittest.TestCase method)
assertIsNone() (unittest.TestCase method)
assertIsNot() (unittest.TestCase method)
assertIsNotNone() (unittest.TestCase method)
assertItemsEqual() (unittest.TestCase method)
assertLess() (unittest.TestCase method)
assertLessEqual() (unittest.TestCase method)
assertListEqual() (unittest.TestCase method)
assertMultiLineEqual() (unittest.TestCase method)
assertNotAlmostEqual() (unittest.TestCase method)
assertNotEqual() (unittest.TestCase method)
assertNotIn() (unittest.TestCase method)
assertNotIsInstance() (unittest.TestCase method)
assertNotRegexpMatches() (unittest.TestCase method)
assertRaises() (unittest.TestCase method)
assertRaisesRegexp() (unittest.TestCase method)
assertRegexpMatches() (unittest.TestCase method)
asserts (2to3 fixer)
assertSequenceEqual() (unittest.TestCase method)
assertSetEqual() (unittest.TestCase method)
addinfo() (hotshot.Profile method)
addition
addLevelName() (in module logging)
addnstr() (curses.window method)
AddPackagePath() (in module modulefinder)
address (multiprocessing.connection.Listener attribute)
  (multiprocessing.managers.BaseManager attribute)
address_family (SocketServer.BaseServer attribute)
address_string()
  (BaseHTTPServer.BaseHTTPRequestHandler method)
AddressList (class in rfc822)
addresslist (rfc822.AddressList attribute)
adssitesdir() (in module site)
addshape() (in module turtle)
addtestsdir() (in module modulefinder)
addSkip() (unittest.TestResult method)
addstr() (curses.window method)
addSuccess() (unittest.TestResult method)
addTests() (unittest.TestSuite method)
addTypeEqualityFunc() (unittest.TestCase method)
addUnexpectedSuccess() (unittest.TestResult method)
adjusted() (decimal.Decimal method)
adler32() (in module zlib)
ADPCM, Intel/DVI
adpcm2lin() (in module audioop)
aepack (module)
AEServer (class in MiniAEFrame)
AEText (class in aetypes)
aetools (module)
aetypes (module)
AF_INET (in module socket)

assertTrue() (unittest.TestCase method)
assertTupleEqual() (unittest.TestCase method)
assignment
  attribute, [1]
augmented
class attribute
class instance attribute
extended slice
slice
slicing
statement, [1]
subscription
subscription
  target list
AST (class in ast)
ast (module)
astimezone() (datetime method)
ASTVisitor (class in compiler.visitor)
async_chat (class in asyncore)
async_chat.ac_in_buffer_size (in module asyncore)
async_chat.ac_out_buffer_size (in module asyncore)
asyncevents() (FrameWork.Application method)
asynchat (module)
asyncore (module)
AsyncResult (class in multiprocessing.pool)
atexit (module)
atime (in module cd)
AF_INET6 (in module socket)
AF_UNIX (in module socket)
aifc (module)
aifc() (aifc.aifc method)
AIFF, [1]
aiff() (aifc.aifc method)
AIFF-C, [1]
AL
   module
AL (module)
al (module)
alarm() (in module signal)
alaw2lin() (in module audioop)
ALERT_DESCRIPTION_HANDSHAKE_FAILURE (in module ssl)
ALERT_DESCRIPTION_INTERNAL_ERROR (in module ssl)
algorithms_available (in module hashlib)
algorithms_guaranteed (in module hashlib)
alignment() (in module ctypes)
all() (built-in function)
all_errors (in module ftplib)
all_features (in module xml.sax.handler)
all_properties (in module xml.sax.handler)
allocate_lock() (in module thread)
allow_reuse_address (SocketServer.BaseServer.attribute)
allowed_domains()
(cookielib.DefaultCookiePolicy method)
alt() (in module curses.ascii)
ALT_DIGITS (in module locale)
altsep (in module os)
altzone (in module time)
ALWAYS_TYPED_ACTIONS (optparse.Option attribute)
AMPER (in module token)
AMPEREQUAL (in module token)

atof() (in module locale)
atoi() (in module locale)
atol() (in module string)
attach() (email.message.Message method)
AttrlistDeclHandler() (xml.parsers.expat.xmlparser method)
attrgetter() (in module xml.etree.ElementTree
        attribute, [1]
        assignment, [1]
        assignment, class
        assignment, class
        class
        class instance
deletion
generic special
reference
special
AttributeError
    exception
attributes (xml.dom.
    AttributesImpl (class
    AttributesNSImpl (class
    attroff() (curses.window
    attron() (curses.window
    attrset() (curses.window
    audio (in module cd)
Audio Interchange File
AUDIO_FILE_ENCODING_ADPCM_G721 (in module sunau)
AUDIO_FILE_ENCODING_ADPCM_G722 (in module sunau)
AUDIO_FILE_ENCODING_ADPCM_G723_3 (in module sunau)
AUDIO_FILE_ENCC
(AUDIO_FILE_ENCC
 module sunau)
AUDIODEV
audioop (module)
auth() (ftplib.FTP_TLS
authenticate() (imaplib.IMAP4
AuthenticationError
authenticators() (netrc
authkey (multiprocessing.
autoGIL (module)
AutoGILError
avg() (in module audio
avgpp() (in module audio
b16decode() (in module base64)
b16encode() (in module base64)
b2a_base64() (in module binascii)
b2a_hex() (in module binascii)
b2a_hqx() (in module binascii)
b2a_qp() (in module binascii)
b2a_uu() (in module binascii)
b32decode() (in module base64)
b32encode() (in module base64)
b64decode() (in module base64)
b64encode() (in module base64)
Babyl (class in mailbox)
BabylMailbox (class in mailbox)
BabylMessage (class in mailbox)
back() (in module turtle)
back quotes, [1]
BACKQUOTE (in module token)
backslash character
backslashreplace_errors() (in module codecs)
backward
bkgdset() (curses.window method)
blank line
block
code
blocked_domains()
(cookielib.DefaultCookiePolicy method)
BlockingIOError
BLOCKSIZE (in module cd)
blocksize (in module sha)
BNF, [1]
body() (ntplib.NNTP method)
body_encode() (email.charset.Charset method)
body_encoding (email.charset.Charset attribute)
body_line_iterator() (in module email.iterators)
BOM (in module codecs)
BOM_BE (in module codecs)
BOM_LE (in module codecs)
BOM_UTF16 (in module codecs)
BOM_UTF16_BE (in module codecs)
BOM_UTF16_LE (in module codecs)
BOM_UTF32 (in module codecs)
BOM_UTF32_BE (in module codecs)
BOM_UTF32_LE (in module codecs)
BOM_UTF8 (in module codecs)
bool (built-in class)

Boolean
    object, [1]
    operation
    operations, [1]
type
values
quotes, [1]
backward() (in module turtle)
backward_compatible (in module imageop)
BadStatusLine
BadZipfile
Balloon (class in Tix)
bare except
base64
  encoding
  module
base64 (module)
BaseCGIHandler (class in wsgiref.handlers)
BaseCookie (class in Cookie)
BaseException
BaseHandler (class in urllib2)
  (class in wsgiref.handlers)
BaseHTTPRequestHandler (class in BaseHTTPServer)
BaseHTTPServer (module)
BaseManager (class in multiprocessing.managers)
  basename() (in module os.path)
BaseProxy (class in multiprocessing.managers)
BaseRequestHandler (class in SocketServer)
BaseServer (class in SocketServer)
basestring (2to3 fixer)
basestring() (built-in function)
basicConfig() (in module
  Boolean (class in aetypes)
boolean() (in module xmlrpclib)
BooleanType (in module types)
bootstrap() (in module ensurepip)
border() (curses.window method)
bottom() (curses.panel.Panel method)
bottom_panel() (in module curses.panel)
BoundaryError
BoundedSemaphore (class in multiprocessing)
BoundedSemaphore() (in module threadir
  (multiprocessing.managers.SyncManager
    method)
box() (curses.window method)
break
  statement, [1], [2], [3], [4]
break_anywhere() (bdb.Bdb method)
break_here() (bdb.Bdb method)
break_long_words (textwrap.TextWrapper
  attribute)
BREAK_LOOP (opcode)
break_on_hyphens (textwrap.TextWrapper
  attribute)
Breakpoint (class in bdb)
breakpoints
BROWSER, [1]
bsddb
  module, [1], [2], [3]
bsddb (module)
BsdDbShelf (class in shelve)
btopen() (in module bsddb)
buffer
  built-in function
    object, [1]
buffer (2to3 fixer)
      (io.TextIOBase attribute)
        (unittest.TestResult attribute)
buffer interface
logging)
BasicContext (class in decimal)
Bastion (module)
Bastion() (in module Bastion)
BastionClass (class in Bastion)
baudrate() (in module curses)
bbox() (ttk.Treeview method)

bdb
    module
Bdb (class in bdb)
bdb (module)
BdbQuit
BDFL
bdist_msi (class in distutils.command.bdist_msi)
beep() (in module curses)
Beep() (in module winsound)
begin_fill() (in module turtle)
begin_poly() (in module turtle)
below() (curses.panel.Panel method)
Benchmarking
benchmarking
betavariate() (in module random)
bgcolor() (in module turtle)
bgn_group() (fl.form method)
bgpic() (in module turtle)
bias() (in module audioop)
bidirectional() (in module unicodedata)
BigEndianStructure (class in ctypes)
bin() (built-in function)

buffer size, I/O
buffer() (built-in function)
buffer_info() (array.array method)
buffer_size (xml.parsers.expat.xmlparser attribute)
buffer_text (xml.parsers.expat.xmlparser attribute)
buffer_used (xml.parsers.expat.xmlparser attribute)
BufferedIOBase (class in io)
BufferedReader (class in io)
BufferedRWPair (class in io)
BufferedWriter (class in io)
BufferError
BufferingHandler (class in logging.handlers)
BufferTooShort
BufferType (in module types), [1]
BUFSIZ (in module macostools)
bufsize() (ossaudiodev.oss_audio_device method)
BUILD_CLASS (opcode)
BUILD_LIST (opcode)
BUILD_MAP (opcode)
build_opener() (in module urllib2)
BUILD_SET (opcode)
BUILD_SLICE (opcode)
BUILD_TUPLE (opcode)
buildtools (module)

built-in

method
abs, [1]
apply, [1], [2]
buffer
bytes
binary
  arithmetic operation
  bitwise operation
  data, packing
Binary (class in msilib)
  (class in xmlrpclib)
binary literal
binary semaphores
BINARY_ADD (opcode)
BINARY_AND (opcode)
BINARY_DIVIDE (opcode)
BINARY_FLOOR_DIVIDE (opcode)
BINARY_LSHIFT (opcode)
BINARY_MODULO (opcode)
BINARY_MULTIPLY (opcode)
BINARY_OR (opcode)
BINARY_POWER (opcode)
BINARY_RSHIFT (opcode)
BINARY_SUBSCR (opcode)
BINARY_SUBTRACT (opcode)
BINARY_TRUE_DIVIDE (opcode)
BINARY_XOR (opcode)
binascii (module)
bind (widgets)
bind() (asyncore.dispatcher method)
  (socket.socket method)
bind_textdomain_codeset() (in module gettext)

binding
  global name
  name, [1], [2], [3], [4], [5]
bindtextdomain() (in module

call
  chr, [1]
  classmethod
cmp, [1], [2], [3]
coerce
compile, [1], [2], [3], [4]
complex, [1]
divmod, [1], [2]
eval, [1], [2], [3], [4], [5], [6], [7]
execfile, [1]
file
  float, [1], [2], [3]
  globals
hash, [1], [2]
help
hex
id
input, [1]
int, [1], [2]
len, [1], [2], [3], [4], [5], [6], [7], [8],
  [10], [11]
locals
long, [1], [2], [3]
max
min
object, [1]
oct
open, [1]
ord, [1], [2]
pow, [1], [2], [3], [4], [5]
range
raw_input, [1]
reload, [1], [2], [3]
repr, [1], [2], [3], [4], [5]
gettext)
binhex
  module
binhex (module)
binhex() (in module binhex)
bisect (module)
bisect() (in module bisect)
bisect_left() (in module bisect)
bisect_right() (in module bisect)
bit_length() (int method)
  (long method)
bitmap() (msilib.Dialog method)
bitwise
  and
    operation, binary
    operation, unary
    operations
  or
    xor
bk() (in module turtle)
bkgd() (curses.window method)
slice, [1], [2]
  staticmethod
str, [1], [2], [3]
tuple, [1]
type, [1], [2], [3]
unichr
unicode, [1], [2], [3]
xrange
built-in method
  call
    object, [1]
builtin_module_names (in module sys)
BuiltinFunctionType (in module types)
BuiltinImporter (class in imputil)
BuiltinMethodType (in module types)
builtins
  module
ButtonBox (class in Tix)
bye() (in module turtle)
byref() (in module ctypes)
byte
byte-code
  file, [1], [2]
byte_compile() (in module distutils.util)
bytearray
  object, [1]
bytearray (built-in class)
bytecode, [1]
byteorder (in module sys)
bytes
  built-in function
bytes (uuid.UUID attribute)
bytes-like object
bytes_le (uuid.UUID attribute)
BytesIO (class in io)
byteswap() (array.array method)
bz2 (module)
BZ2Compressor (class in bz2)
BZ2Decompressor (class in bz2)
BZ2File (class in bz2)
C

language, [1], [2], [3], [4]
structures

c_bool (class in ctypes)
C_BUILTIN (in module imp)
c_byte (class in ctypes)
c_char (class in ctypes)
c_char_p (class in ctypes)
c_double (class in ctypes)
C_EXTENSION (in module imp)
c_float (class in ctypes)
c_int (class in ctypes)
c_int16 (class in ctypes)
c_int32 (class in ctypes)
c_int64 (class in ctypes)
c_int8 (class in ctypes)
c_long (class in ctypes)
c_longdouble (class in ctypes)
c_longlong (class in ctypes)
c_short (class in ctypes)
c_size_t (class in ctypes)
c_ssize_t (class in ctypes)
c_ubyte (class in ctypes)
c_uint (class in ctypes)
c_uint16 (class in ctypes)
c_uint32 (class in ctypes)
c_uint64 (class in ctypes)
c_uint8 (class in ctypes)
c_ulong (class in ctypes)
c_ulonglong (class in ctypes)
c_ushort (class in ctypes)
c_void_p (class in ctypes)
c_wchar (class in ctypes)
c_wchar_p (class in ctypes)
CAB (class in msilib)

code object, [1], [2], [3]
code() (in module new)
Codecs
decode
encode
codecs (module)
coded_value (Cookie.Morsel attribute)
encode (module)
coerce()
coerce() (built-in function)
coercion

col_offset (ast.AST attribute)
collapse_rfc2231_value(email.utils)
collect() (in module gc)
collect Incoming_data() (method)
collections (module)
COLON (in module token)
color() (in module fl)
ColorPicker (module)
colorsys (module)
color_content() (in module curses)
color_pair() (in module curses)
colormode() (in module turtle)
COLUMNS
combinations() (in module itertools)
CacheFTPHandler (class in urllib2)
calcsze() (in module struct)
Calendar (class in calendar)
calendar (module)
calendar() (in module calendar)
call
  built-in function
  built-in method
  class instance
  class object, [1], [2], [3]
  function, [1], [2]
  instance, [1]
  method
  procedure
  user-defined function
call() (dl.dl method)
  (in module subprocess)
CALL_FUNCTION (opcode)
CALL_FUNCTION_KW (opcode)
CALL_FUNCTION_VAR (opcode)
CALL_FUNCTION_VAR_KW (opcode)
call_tracing() (in module sys)
callable
  object, [1]
callable (2to3 fixer)
Callable (class in collections)
callable() (built-in function)
CallableProxyType (in module weakref)
callback (optparse.Option attribute)
callback() (MiniAEFrame.AEServer method)
callback_args (optparse.Option attribute)
callback_keywords (optparse.Option attribute)
CalledProcessError
calloc()
can_change_color() (in module curses)
can_fetch() (robotparser.RobotFileParser method)
combinations_with_replacement (itertools)
combine() (datetime.datetime method)
combining() (in module unicodedata)
ComboBox (class in Tix)
Combobox (class in ttk)
comma
  trailing, [1]
COMMA (in module tokenize)
command
  command
command_line
  command line
command_line_option
    --help
    --version
    -3
    -?
    -B
    -E
    -J
    -O
    -OO
    -Q <arg>
    -R
    -S
    -U
    -V
    -W arg
    -X
    -c <command>
    -d
    -h
    -i
cancel() (sched.scheduler method)
    (threading.Timer method)
cancel_join_thread() (multiprocessing.Queue method)
CannotSendHeader
CannotSendRequest
canonic() (bdb.Bdb method)
canonical() (decimal.Context method)
    (decimal.Decimal method)
capitalize() (in module string)
    (str method)
Capsule
    object
captured_stdout() (in module test.test_support)
captureWarnings() (in module logging)
capwords() (in module string)
Carbon.AE (module)
Carbon.AH (module)
Carbon.App (module)
Carbon.Appearance (module)
Carbon.CarbonEvents (module)
Carbon.CarbonEvt (module)
Carbon_CF (module)
Carbon.CG (module)
Carbon.Cm (module)
Carbon.Components (module)
Carbon.ControlAccessor (module)
Carbon.Controls (module)
Carbon.CoreFounation (module)
Carbon.CoreGraphics (module)
Carbon.CtI (module)
Carbon.Dialogs (module)
Carbon.Dlg (module)
Carbon.Drag (module)
Carbon.Dragconst (module)
Carbon.Events (module)
CarbonEvt (module)

-m <module-name>
-s
-t
-u
-v
-x
CommandCompiler (class commands (module)
comment
(cookielib.Cookie attribute)
COMMENT (in module t
comment (zipfile.ZipFile
    (zipfile.ZipInfo attribute)
Comment() (in module xml.etree.ElementTree)
comment_url (cookielib.Cookie attribute)
commenters (shlex.shlex attribute)
CommentHandler() (xml.parsers.expat.xmlp
commit() (msilib.CAB method)
Commit() (msilib.Database method)
commit() (sqlite3.Connection method)
common (filecmp.dircmp attribute)
Common Gateway Interface
common_dirs (filecmp.dircmp attribute)
common_files (filecmp.dircmp attribute)
common_funny (filecmp.dircmp attribute)
common_types (in module mimetypes)
commonprefix() (in module os.path)
communicate() (subprocess.Popen method)
compare_digest() (in module hmac)
COMPARE_OP (opcode)
compare_signal() (decimal.Decimal method)
catalog (in module cd)
catch_warnings (class in warnings)
category() (in module unicodedata)
cbreak() (in module curses)
CC
CCompiler (class in distutils.ccompiler)
cd (module)
CDLL (class in ctypes)
CDROM (in module cd)
ceil() (in module math), [1]
center() (in module string)
    (str method)
CERT_NONE (in module ssl)
CERT_OPTIONAL (in module ssl)
CERT_REQUIRED (in module ssl)
cert_store_stats() (ssl.SSLContext method)
cert_time_to_seconds() (in module ssl)
CertificateError
certificates
CFLAGS, [1], [2]
cfmfile (module)
CFUNCTYPE() (in module ctypes)
CGI
    debugging
    exceptions
    protocol
    security
    tracebacks
cgi (module)
cgi_directories
(CGIHTTPRequestHandler attribute)
CGIHandler (class in wsgiref.handlers)
CGIHTTPRequestHandler (class in CGIHTTPServer)
CGIHTTPServer
    module
CGIHTTPServer (module)
test.test_support)
checkbox() (msilib.Dialog method)
checkcache() (in module linecache)
checkfuncname() (in module bdb)
CheckList (class in Tix)
checksum
    Cyclic Redundancy Check
    MD5
    SHA
chflags() (in module os)
chdir (curses.window method)
childerr (popen2.Popen3 attribute)
childNodes (xml.dom.Node attribute)
chmod() (in module os)
choice() (in module random)
choices (optparse.Option attribute)
choose_boundary() (in module mimetools)
chown() (in module os)
chr
    built-in function, [1]
chr() (built-in function)
chroot() (in module os)
Chunk (class in chunk)
chunk (module)
cipher
    DES
cipher() (ssl.SSLSocket method)
circle() (in module turtle)
CIRCUMFLEX (in module token)
CIRCUMFLEXEQUAL (in module token)
Clamped (class in decimal)
class
    attribute
    attribute assignment
    classic
    constructor
    definition, [1]
    (smtplib.SMTP method)
    (socket.socket method)
    connect_ex() (socket.socket method)
    Connection (class in multiprocessing)
    (class in sqlite3)
    connection (sqlite3.Cursor)
    ConnectRegistry () (in module _winreg)
    const (optparse.Option attribute)
    constructor
class
    constructor() (in module container, [1]
    iteration over
    Container (class in container)
    contains() (in module operator)
content type
    MIME
    ContentHandler (class in xml.sax)
    Context (class in decimal)
    context (ssl.SSLSocket)
    context management protocol
    context_manager, [1], [2]
    context_diff() (in module contextlib)
    contextmanager() (in module contextlib)
    continue
    statement, [1], [2], [3]
    CONTINUE_LOOP (opcode)
    Control (class in msilib)
    (class in Tix)
control (in module cd)
control() (msilib.Dialog method)
    (select.kqueue method)
controlnames (in module...
instance
name
new-style
object, [1], [2], [3], [4]
old-style
statement
Class (class in symtable)
Class browser
class instance
  attribute
  attribute assignment
  call
  object, [1], [2], [3]
class object
  call, [1], [2], [3]
classic class
classmethod
  built-in function
  classmethod() (built-in function)
classobj() (in module new)
ClassType (in module types), [1]
class
clause
clean() (mailbox.Maildir method)
cleandoc() (in module inspect)
cleanup functions
Clear Breakpoint
clear() (collections.deque method)
  (cookielib.CookieJar method)
  (curses.window method)
  (dict method)
  (in module turtle), [1]
  (mailbox.Mailbox method)
  (set method)
  (threading.Event method)
  (xml.etree.ElementTree.Element method)
clear_all_breaks() (bdb.Bdb method)
controls() (ossaudiodev.method)
conversion
  arithmetic
  string, [1], [2]
ConversionError
conversions
  numeric
  convert() (email.charset.
    convert_arg_line_to_arg
    (argparse.ArgumentParser
    convert_charref() (sgml.methods)
    convert_codepoint() (sgml
    convert_entityref() (sgml
    convert_field() (string.Formatter
    convert_path() (in module
class Cookie (class in cookielib
  (module)
   CookieError
   CookieJar (class in cooki
   cookiejar (urllib2.HTTPC
   attribute)
   cookielib (module)
   CookiePolicy (class in co
   Coordinated Universal T
   Copy
   copy
  module
   copy (module)
   copy() (decimal.Context
  (dict method)
   (hashlib.hash method)
   (hmac.HMAC method)
   (imaplib.IMAP4 meth
(ftplib.FTP method)
(generator method)
(hotshot.Profile method)
(httplib.HTTPConnection method)
(imaplib.IMAP4 method)
(in module anydbm)
(in module dbm)
(in module dumbdbm)
(in module fileinput)
(in module gdbm)
(in module os), [1]
(io.IOBase method)
(logging.FileHandler method)
(logging.Handler method)
(logging.handlers.MemoryHandler method)
(logging.handlers.NTEventLogHandler method)
(logging.handlers.SocketHandler method)
(logging.handlers.SysLogHandler method)
(mailbox.MH method)
(mailbox.Mailbox method)
(mailbox.Maildir method)
(mmap.mmap method)
Close() (msilib.View method)
close() (multiprocessing.Connection method)
(StringIO.StringIO method)
(multiprocessing.Queue method)
(multiprocessing.connection.Listener method)
(multiprocessing.pool.multiprocessing.Pool method)
(ossaudiodev.oss_audio_device method)
(ossaudiodev.oss_mixer_device method)
(select.epoll method)
cos() (in module cmath)
(in module math)
cosh() (in module cmath)
(in module math)
count() (array.array method)
(collections.deque method)
(in module itertools)
(in module string)
(str method)
Counter (class in collections)
countOf() (in module operator)
countTestCases() (unittest.TestCase method)
(CoverageResults class in unitest.TestSuite method)
cPickle module
cPickle (module)
CPP
CPPFLAGS
cProfile (module)
CPU time
cpu_count() (in module multiprocessing)
CPython
CRC (zipfile.ZipInfo attribute)
crc32() (in module binascii)
(in module zlib)
crc_hqx() (in module binascii)
create() (imaplib.IMAP4 method)
create_aggregate() (sqlite3.Connection method)
create_collation() (sqlite3.Connection method)
create_connection() (sqlite3.Connection method)
create_decimal() (decimal.Context method)
create_decimal_from_float() (decimal.Context method)
create_default_context()
cmdloop() (cmd.Cmd method)
cmdqueue (cmd.Cmd attribute)
cmp
    built-in function, [1], [2], [3]
cmp() (built-in function)
    (in module filecmp)
cmp_op (in module dis)
cmp_to_key() (in module functools)
cmpfiles() (in module filecmp)
co_argcount (code object attribute)
co_cellvars (code object attribute)
co_code (code object attribute)
co_consts (code object attribute)
co_filename (code object attribute)
co_firstlineno (code object attribute)
co_flags (code object attribute)
co_freevars (code object attribute)
CO_FUTURE_DIVISION (C variable)
co_lnotab (code object attribute)
co_name (code object attribute)
co_names (code object attribute)
co_nlocals (code object attribute)
co_stacksize (code object attribute)
co_varnames (code object attribute)
CO蓼底蓼底
object
code
    block
code (module)
    (urllib2.HTTPError attribute)
    (xml.parsers.expat.ExpatError attribute)
method)
credits (built-in variable)
critical() (in module logging)
    (logging.Logger method)
CRNCYSTR (in module xml)
crypt() (in module image)
crypt(3), [1], [2]
cryptography
CStringIO (module)
csv
    (module)
ctermid() (in module os)
cftime() (datetime.date method)
    (datetime.datetime method)
    (in module time)
ctrl() (in module curses.ascii)
CTRL_BREAK_EVENT
CTRL_C_EVENT (in module signal)
crypt()
(crypt (module)
crypt(3), [1], [2]
cryptography
currency()
(ttk.Combobox method)
current() (in module multiprocessing)
current_process() (in module threading)
current_thread() (in module threading)
CurrentByteIndex
(xml.parsers.expat.xmlp
CurrentLineNumber
(xml.parsers.expat.xmlp
CurrentThread()
(xml.parsers.expat.xmlp
currentThread() (in module
CObject
    object
class
    block
class (module)
    (urllib2.HTTPError attribute)
    (xml.parsers.expat.ExpatError attribute)
curses (module)
curses.ascii (module)
curses.panel (module)
curses.textpad (module)
Cursor (class in sqlite3)
cursor() (sqlite3.Connection method)
cursyncup() (curses.window method)
curval (EasyDialogs.ProgressBar attribute)
customize_compiler() (in module distutils.sysconfig)
Cut
cwd() (ftpplib.FTP method)
cycle() (in module itertools)
Cyclic Redundancy Check
D

D_FMT (in module locale)
D_T_FMT (in module locale)
daemon (multiprocessing.Process attribute)
    (threading.Thread attribute)
dangling
data
    packing binary
tabular
type
type, immutable
Data (class in plistlib)
data (select.kevent attribute)
    (UserDict.IterableUserDict attribute)
    (UserList.UserList attribute)
    (UserString.MutableString attribute)
    (xml.dom.Comment attribute)
    (xml.dom.ProcessingInstruction attribute)
    (xml.dom.Text attribute)
    (xmlrpcclient.Binary attribute)
data() (xml.etree.ElementTree.TreeBuilder
    method)
database
    Unicode
databases
DatagramHandler (class in logging.handlers)
DatagramRequestHandler (class in SocketServer)
DATASIZE (in module cd)
date (class in datetime)
date() (datetime.datetime method)
    (nnntplib.NNTP method)
date_time (zipfile.ZipInfo attribute)
directory ...
    compileall command
directory_created() (built-in function)
DirList (class in Tix)
dirname() (in module os.
DirSelectBox (class in Tix)
DirSelectDialog (class in Tix)
DirTree (class in Tix)
dis (module)
dis() (in module dis)
    (in module pickletools)
disable() (bdb.Breakpoint
    (in module gc)
    (in module logging)
    (profile.Profile method)
disable_interspersed_args() (optparse.OptionParser
DisableReflectionKey() (in module _winreg)
disassemble() (in module dis)
discover() (unittest.TestLoader
dispatch() (compiler.visitor.ASTVisitor
    dispatch_call() (bdb.Bdb
    dispatch_exception() (bdb.Bdb
    dispatch_line() (bdb.Bdb
    dispatch_return() (bdb.Bdb
    dispatcher (class in asyncio
    dispatcher_with_send (c
    display
method)
DEBUG_SAVEALL (in module gc)
debug_src() (in module doctest)
DEBUG_STATS (in module gc)
DEBUG_UNCOLLECTABLE (in module gc)
debugger, [1], [2]
    configuration file
debugging
    CGI
    assertions
DebuggingServer (class in smtpd)
DebugRunner (class in doctest)
DebugStr() (in module MacOS)
Decimal (class in decimal)
decimal (module)
decimal literal
decimal() (in module unicodedata)
DecimalException (class in decimal)
decode
    Codecs
dedecode() (codecs.Codec method)
        (codecs.IncrementalDecoder method)
        (in module base64)
        (in module codecs)
        (in module mimetools)
        (in module quopri)
        (in module uu)
        (json.JSONDecoder method)
        (str method)
        (xmlrpclib.Binary method)
        (xmlrpclib.DateTime method)
de decode_header() (in module email.header)
de code_params() (in module email.utils)
de code_rfc2231() (in module email.utils)
DecodedGenerator (class in email.generator)
de decodestring() (in module base64)
    (in module quopri)
distutils.debug (module)
distutils.dep_util (module)
distutils.dir_util (module)
distutils.dist (module)
distutils.emxccompiler (module)
distutils.errors (module)
distutils.extension (module)
distutils.fancy_getopt (module)
distutils.file_util (module)
distutils.filelist (module)
distutils.log (module)
distutils.msvcompiler (module)
distutils.spawn (module)
distutils.sysconfig (module)
distutils.text_file (module)
distutils.unixccompiler (module)
distutils.util (module)
distutils.version (module)
DISTUTILS_DEBUG
dither2grey2() (in module
    dither2mono() (in module
    div() (in module operator)
divide() (decimal.Context method)
divide_int() (decimal.Context method)
    integer
    long integer
DivisionByZero (class in
    divmod
    built-in function, [1], [2]
divmod() (built-in function)
    (decimal.Context method)
    DllCanUnloadNow() (in module
    DllGetClassObject() (in module
    dllhandle (in module sys
    dngettext() (in module sys
    do_activate() (FrameWo
decomposition() (in module unicodedata)
decompress() (bz2.BZ2Decompressor method)
    (in module bz2)
    (in module jpeg)
    (in module zlib)
    (zlib.Decompress method)
decompressobj() (in module zlib)
def
  statement
def_prog_mode() (in module curses)
def_shell_mode() (in module curses)
default
  parameter value
default (optparse.Option attribute)
default() (cmd.Cmd method)
    (compiler.visitor.ASTVisitor method)
    (json.JSONEncoder method)
DEFAULT_BUFFER_SIZE (in module io)
default_bufsize (in module xml.dom.pulldom)
default_factory (collections.defaultdict attribute)
DEFAULT_FORMAT (in module tarfile)
default_open() (urllib2.BaseHandler method)
default_timer() (in module timeit)
DefaultContext (class in decimal)
DefaultCookiePolicy (class in cookielib)
defaultdict (class in collections)
DefaultHandler() (xml.parsers.expat.xmlparser method)
DefaultHandlerExpand()
(xml.parsers.expat.xmlparser method)
defaults() (ConfigParser.RawConfigParser method)
  (FrameWork.Window
do_char() (FrameWork.
do_clear() (bdb.Bdb method)
do_command() (curses.
do_contentclick() (Frame
do_controlhit() (FrameW
  (FrameWork.Scroll
do_dialogevent() (Frame method)
do_forms() (in module fl'
do_GET() (SimpleHTTPServer.
do_handshake() (ssl.SS
do_HEAD() (SimpleHTTPServer.
do_itemhit() (FrameW
do_POST() (CGIHTTPServer.
do_postresize() (Frame\
do_update() (FrameW
do_update() (FrameW
docmd() (smtplib.SMTP method)
docstring, [1]
docstrings, [1]
DocTest (class in doctes
doctest (module)
method)
defaultTestLoader (in module unittest)
defaultTestResult() (unittest.TestCase method)
defects (email.message.Message attribute)
define_macro() (distutils.ccompiler.CCompiler method)
definition
class, [1]
function, [1]
defpath (in module os)
degrees() (in module math)
   (in module turtle)
del
   statement, [1], [2], [3]
def_param() (email.message.Message method)
delattr() (built-in function)
delay() (in module turtle)
delay_output() (in module curses)
delayload (cookielib.FileCookieJar attribute)
delch() (curses.window method)
dele() (poplib.POP3 method)
delete() (ftplib.FTP method)
   (imaplib.IMAP4 method)
   (ttk.Treeview method)
DELETE_ATTR (opcode)
DELETE_FAST (opcode)
DELETE_GLOBAL (opcode)
DELETE_NAME (opcode)
DELETE_SLICE+0 (opcode)
DELETE_SLICE+1 (opcode)
DELETE_SLICE+2 (opcode)
DELETE_SLICE+3 (opcode)
DELETE_SUBSCR (opcode)
deleteacl() (imaplib.IMAP4 method)
deletefilehandler() (Tkinter.Widget.tk method)
deletefolder() (mhlib.MH method)
DocTestFailure
DocTestFinder (class in DocTestParser (class in DocTestRunner (class in DocTestSuite() (in module doctype() (xml.etree.ElementTree method)
   (xml.etree.ElementTree)
documentation
generation
   online
documentation string
documentation strings, [
documentElement (xml.etree.ElementTree)
DocXMLRPCRequestHandler (class in DocXMLRPCServer)
DocXMLRPCServer (class in module)
domain_initial_dot (cookielib.Cookie attribute)
domain_return_ok() (cookielib.CookiePolicy method)
domain_specified (cookielib.DefaultCookiePolicy attribute)
DomainRFC2965Match (cookielib.DefaultCookie attribute)
DomainStrict (cookielib.DefaultCookie attribute)
DomainStrictNoDots (cookielib.DefaultCookie attribute)
DomainStrictNonDomain (cookielib.DefaultCookie attribute)
DOMEventStream (class)
DOMException
domstringSizeErr
done() (in module turtle)
   (xdrlib.Unpacker method)
DeleteKey() (in module _winreg)
DeleteKeyEx() (in module _winreg)
deleteln() (curses.window method)
deleteMe() (bdb.Breakpoint method)
DeleteValue() (in module _winreg)
deletion
  attribute
target
target list
delimiter (csv.Dialect attribute)
delimiters
delitem() (in module operator)
deliver_challenge() (in module multiprocessing.connection)
delslice() (in module operator)
demo_app() (in module wsgiref.simple_server)
denominator (numbers.Rational attribute)
DeprecationWarning
deque (class in collections)
DER_cert_to_PEM_cert() (in module ssl)
derwin() (curses.window method)
DES
  cipher
description (sqlite3.Cursor attribute)
description() (ntplib.NNTP method)
descriptions() (ntplib.NNTP method)
descriptor
  file
dest (optparse.Option attribute)
destructor, [1]
Detach() (_winreg.PyHKEY method)
detach() (io.BufferedIOBase method)
  (io.TextIOBase method)
  (ttk.Treeview method)
detect_language()
  (distutils.ccompiler.CCompiler method)
deterministic profiling

DONT_ACCEPT_BLANKLINE (doctest)
DONT_ACCEPT_TRUE_DO (doctest)
dont_write_bytecode (in doRollover)
(logging.handlers.RotatingFileHandler)
  (logging.handlers.TimedRotatingFileHandler)
DOT (in module token)
dot() (in module turtle)
DOTALL (in module re)
doublequote (csv.Dialect)
DOUBLESLASH (in module)
DOUBLESLASHEQUAL (in module)
DOUBLESTAR (in module)
DOUBLESTAREQUAL (in module)
doupdate() (in module curses)
down() (in module turtle)
drop_whitespace (textwrap attribute)
dropwhile() (in module itertools)
dst() (datetime.datetime method)
  (datetime.time method)
  (datetime.tzinfo method)
DTDHandler (class in xml)
duck-typing
dumbdbm
  module
dumbdbm (module)
DumbWriter (class in for
  dummy_thread (module)
dummy_threading (module)
dump() (in module ast)
  (in module json)
  (in module marshal)
  (in module pickle)
DEVICE (module)
devnull (in module os)
dgettext() (in module gettext)
Dialect (class in csv)
dialect (csv.csvreader attribute)
    (csv.csvwriter attribute)
Dialog (class in msilib)
DialogWindow() (in module FrameWork)
dict (2to3 fixer)
    (built-in class)
dict()
    (multiprocessing.managers.SyncManager method)
dictConfig() (in module logging.config)

**dictionary**
    display
        object, [1], [2], [3], [4], [5], [6], [7]
        type, operations on

**dictionary view**
DictionaryType (in module types), [1]
DictMixin (class in UserDict)
DictProxyType (in module types)
DictReader (class in csv)
DictType (in module types), [1]
DictWriter (class in csv)
diff_files (filecmp.dircmp attribute)
Differ (class in difflib), [1]
difference() (set method)
difference_update() (set method)
difflib (module)
digest() (hashlib.hash method)
    (hmac.HMAC method)
    (md5.md5 method)
    (sha.sha method)
digest_size (in module md5)
    (in module sha)
digit() (in module unicodedata)

(in module xml.etree.
    (pickle.Pickler method
dump_address_pair() (in
dump_stats() (profile.Pro
    (pstats.Stats method
dumps() (in module json
    (in module marshal)
    (in module pickle)
    (in module xmlrpclib)
dup() (in module os)
    (posixfile.posixfile method
dup2() (in module os)
    (posixfile.posixfile method
DUP_TOP (opcode)
DUP_TOPX (opcode)
DuplicateSectionError
dwFlags (subprocess.S
DynLoadSuffixImporter (}
digits (in module string)
dir() (built-in function)
   (ftplib.FTP method)
dircache (module)
dircmp (class in filecmp)
directory
   changing
   creating
   deleting, [1]
site-packages
site-python
traversal
walking
Directory (class in msilib)
e (in module cmath)
    (in module math)
E2BIG (in module errno)
EACCES (in module errno)
EADDRINUSE (in module errno)
EADDRNOTAVAIL (in module errno)
EADV (in module errno)
EAFNOSUPPORT (in module errno)
EAFP
EAGAIN (in module errno)
EALREADY (in module errno)
east_asian_width() (in module unicodedata)
EasyDialogs (module)
EBADE (in module errno)
EBADF (in module errno)
EBADFD (in module errno)
EBADMSG (in module errno)
EBADR (in module errno)
EBADRQC (in module errno)
EBADSLT (in module errno)
EBCDIC
EBFONT (in module errno)
EBUSY (in module errno)
ECHILD (in module errno)
echo() (in module curses)
echochar() (curses.window method)
ECHRNG (in module errno)
ECOMM (in module errno)
ECONNABORTED (in module errno)
ECONNREFUSED (in module errno)
ECONNRESET (in module errno)
EDEADLK (in module errno)
EDEADLOCK (in module errno)
EDESTADDRREQ (in module errno)
edit() (curses.textpad.Textbox method)

EPFNOSUPPORT
epilogue (email.message.Message attribute)
EPIPE (in module select)
epoch
epoll() (in module select)
EPROTO (in module socket)
EPROTONOSUPPORT
EPROTOTYPE
eq() (in module operator)
EQEQUAL (in module token)
EQUAL (in module token)
ERA (in module locale)
ERA_D_FMT
ERA_D_T_FMT
ERA_T_FMT
ERANGE (in module math)
erase() (curses.window method)
erasechar() (in module curses)
EREMCHG (in module errno)
EREMOTE (in module errno)
EREMOTEIO
ERESTART (in module select)
erf() (in module math)
erfc() (in module math)
EROFS (in module errno)
ERR (in module curses)
errcheck (ctypes._FuncPtr attribute)
errno
errmsg (xmlrpc.client.ProtocolError attribute)
epoch
elif  # keyword

...
Ellinghouse, Lance

Ellipsis
  object
Ellipsis (built-in variable)
ELLIPSIS (in module doctest)
EllipsisType (in module types)
ELNRNG (in module errno)
ELOOP (in module errno)
else
dangling
  keyword, [1], [2], [3], [4]
e-mail (module)
e-mail.charset (module)
e-mail.encoders (module)
e-mail.errors (module)
e-mail.generator (module)
e-mail.header (module)
e-mail.iterators (module)
e-mail.message (module)
e-mail.mime (module)
e-mail.parser (module)
e-mail.utils (module)
EMFILE (in module errno)
emit() (logging.FileHandler method)
  (logging.Handler method)
  (logging.NullHandler method)
  (logging.StreamHandler method)
  (logging.handlers.BufferingHandler method)
  (logging.handlers.DatagramHandler method)
  (logging.handlers.HTTPHandler method)
  (logging.handlers.NTEventLogHandler method)
  (logging.handlers.RotatingFileHandler method)
  (logging.handlers.SMTPHandler method)
  (logging.handlers.SocketHandler method)
  (logging.handlers.SysLogHandler method)
  (logging.handlers.TimedRotatingFileHandler method)
  (logging.handlers.WatchedFileHandler method)
eval
attribute)
Errors
  logging
  errors
    (file attribute)
    (io.TextIOBase attribute)
    (unittest.TestCase attribute)
ErrorString() (in module xml.parsers.expat)
ERRORTOKEN (in module token)
escape (shlex) escape sequence escape() (in module)
  (in module)
  (in module)
  escapedchars (attribute)
ESHUTDOWN (in module)
ESOCKTNOSUPPORT (in module errno)
ESPIPE (in module errno)
ESRCH (in module errno)
ESRMNT (in module errno)
ESTALE (in module errno)
ESRIPipe (in module)
ETIME (in module errno)
ETIMEDOUT (in module errno)
Etiny() (decimal.Context method)
ETIMEDOUT (in module)
Eval (class in multiprocessing) built-in function eval(expr)
Eval () (built-in function) built-in evaluation
Event (class in multiprocessing)
**EMLINK** (in module errno)

Empty

def empty()
    list
tuple, [1]

empty()
    (multiprocessing.multiprocessing.queues.SimpleQueue method)
    (Queue.Queue method)
    (multiprocessing.Queue method)
    (sched.scheduer method)

**EMPTY_NAMESPACE** (in module xml.dom)

emptyline() (cmd.Cmd method)

**EMSGSIZE** (in module errno)

**EMULTIHOP** (in module errno)

enable() (bdb.Breakpoint method)
    (in module cgitb)
    (in module gc)
    (profile.Profile method)

enable_callback_tracebacks() (in module sqlite3)

enable_interspersed_args() (optparse.OptionParser method)

enable_load_extension() (sqlite3.Connection method)

**ENABLE_USER_SITE** (in module site)

EnableReflectionKey() (in module _winreg)

**ENAMETOOLONG** (in module errno)

**ENCLOSE** (in module errno)

**ENCLOSE** (in module curses)
enclose() (curses.window method)

**encode**
    Codecs
    encode() (codecs.Codec method)
    (codecs.IncrementalEncoder method)
    (email.header.Header method)
    (in module base64)
    (in module codecs)
    (in module mimetools)

(event scheduling)

**event()** (msilib.Control method)

**Event()** (multiprocessing.managers.SyncManager method)

**events** (widgets)

**EWOULDexampleModalLabel**

**EX_CANTCREAT** (in module os)

**EX_CONFIG** (in module os)

**EX_DATAERR** (in module os)

**EX_IOERR** (in module os)

**EX_NOHOST** (in module os)

**EX_NOTFOUND** (in module os)

**EX_NOUSER** (in module os)

**EX_OK** (in module os)

**EX_OSERR** (in module os)

**EX_OSFILE** (in module os)

**EX_PROTOCOL** (in module os)

**EX_SOFTWARE** (in module os)

**EX_TEMPFAIL** (in module os)

**EX_UNAVAILABLE** (in module os)

**EX_USAGE** (in module sys)

**Example**

**exception**

**exc_clear()** (in module sys)

**exc_info()** (in module sys)

**exc_info()** (in module doc)

**exc_info()** (in module excjson)

**exc_msg** (doctest)

**exc_traceback** (in module sys)

**exc_type** (in module sys)

**exc_value** (in module sys)

**excel** (class in module csv)

**excel_tab** (class in module csv)
except
    bare
    keyword
    statement
except (2to3 fix
excepthook()
Exception
exception, [1]
    AssertionError
    AttributeError
    Generator
    ImportError
    NameError
    RuntimeError
    StopIteration
    TypeError
    ValueError
    ZeroDivisionError
    handler
    raising
EXCEPTION
except handler
exception() (in
    (logging.Logger	method)
exceptions
    in CGI scr
module
exceptions (m exclusive
    or
EXDEV (in m
exec
    statement
exec (2to3 fix exec_prefix, [ EXEC_PREF
END FINALLY (opcode)
end_group() (fl.form method)
end_headers()
(BaseHTTPServer.BaseHTTPRequestHandler method)
end_marker() (multifile.MultiFile method)
end_paragraph() (formatter.formatter method)
end_poly() (in module turtle)
EndCdataSectionHandler() (xml.parsers.expat.xmlparser method)
EndDoctypeDeclHandler() (xml.parsers.expat.xmlparser method)
endDocument() (xml.sax.handler.ContentHandler method)
endElement() (xml.sax.handler.ContentHandler method)
EndElementHandler() (xml.parsers.expat.xmlparser method)
endElementNS() (xml.sax.handler.ContentHandler method)
endheaders() (httplib.HTTPConnection method)
ENDMARKER (in module token)
EndNamespaceDeclHandler()
(xml.parsers.expat.xmlparser method)
endpick() (in module gl)
endpos (re.MatchObject attribute)
endPrefixMapping() (xml.sax.handler.ContentHandler method)
endselect() (in module gl)
endswith() (str method)
endwin() (in module curses)
ENETDOWN (in module errno)
ENETRESET (in module errno)
ENETUNREACH (in module errno)
ENFILE (in module errno)
ENOANO (in module errno)
ENOBUFS (in module errno)
ENOCSSI (in module errno)
ENODATA (in module errno)
ENODEV (in module errno)
ENOENT (in module errno)
exec_prefix (in module sys)
EXEC_STMT (opcode)
execfile (built-in function)
execfile (2to3 fixer)
execfile (built-in function)
excl() (in module os)
execle() (in module os)
execp() (in module os)
exclpe() (in module os)
executable (in module os)
executable_file (distutils.command)
execute() (distribution)
Execute() (msilib.View method)
execute() (sqlite3.Connection method)
exectumany (sqlite3.Cursor method)
execturescript (sqlite3.Cursor method)
execution frame, [1]
restricted stack
execution model
execv() (in module os)
evcve() (in module os)
excvp() (in module os)
evcvp() (in module os)
ExFileSelectBox (class in Tix)
EXFULL (in module errno)
exists() (in module os)
(ttk.Treeview method)
exit (built-in variable)
exit()
ENOEXEC (in module errno)  
ENOLCK (in module errno)  
ENOLINK (in module errno)  
ENOMEM (in module errno)  
ENOMSG (in module errno)  
ENONET (in module errno)  
ENOPKG (in module errno)  
ENOPROTOOPT (in module errno)  
ENOSPC (in module errno)  
ENOSR (in module errno)  
ENOSTR (in module errno)  
ENOSYS (in module errno)  
ENOTBLK (in module errno)  
ENOTCONN (in module errno)  
ENOTDIR (in module errno)  
ENOTEMPTY (in module errno)  
ENOTNAM (in module errno)  
ENOTSOCK (in module errno)  
ENOTTY (in module errno)  
ENOTUNIQ (in module errno)  
enurepip (module)  
enter() (sched.scheduler method)  
enterabs() (sched.scheduler method)  
entities (xml.dom.DocumentType attribute)  
EntityDeclHandler() (xml.parsers.expat.xmlparser method)  
entitydefs (in module htmlentitydefs)  
EntityResolver (class in xml.sax.handler)  
Enum (class in aetypes)  
enum_certificates() (in module ssl)  
enum_crls() (in module ssl)  
enumerate() (built-in function)  
  (in module fm)  
  (in module threading)  
EnumKey() (in module _winreg)  
enumsubst() (in module aetools)  
EnumValue() (in module _winreg)  
environ (in module os)  
(argparse.ArgumentParser method)  
(in module sys)  
exitonclick() (in module turtle)  
exp() (decimal.Decimal method)  
(expectedFailures (unittest.TestResult attribute)  
ExpatError  
expect() (telnetlib.Telnet method)  
extendedFailures  
expires (cookielib.Cookie attribute)  
expm1() (in math)  
expovariate() (in module random)  
expression, [1]  
Condition:  
conditiona  
generator  
lambda, [1]  
list, [1], [2]  
statement
environment

environment variable

%PATH%

<protocol>_proxy

APPDATA

AUDIODEV

BROWSER, [1]

CC

CFLAGS, [1], [2]

COLUMNS, [1]

COMSPEC, [1]

CPP

CPPFLAGS

DISTUTILS_DEBUG

HOME, [1], [2], [3], [4], [5]

HOMEDRIVE, [1]

HOMEPATH, [1]

IDLESTARTUP

KDEDIR

LANG, [1], [2], [3], [4]

LANGUAGE, [1]

LC_ALL, [1]

LC_MESSAGES, [1]

LDCXXSHARED

LDFLAGS

LINES, [1], [2]

LNAME

LOGNAME, [1]

MIXERDEV

PAGER, [1]

PATH, [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18]

PLAT
POSIXLY_CORRECT
PYTHON*
PYTHONCASEOK, [1]
PYTHONDEBUG, [1]
PYTHONDOCS
PYTHONDONTWRITEBYTECODE, [1], [2], [3], [4]
PYTHONDUMPREFS, [1]
PYTHONEXECUTABLE
PYTHONHASHSEED, [1], [2], [3]
PYTHONHOME, [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]
PYTHONHTTPSVERIFY
PYTHONINSPECT, [1], [2]
PYTHONIOENCODING, [1]
PYTHONMALLOCSTATS
PYTHONNOUSERSITE, [1], [2], [3]
PYTHONOPTIMIZE, [1]
PYTHONPATH, [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]
PYTHONSTARTUP, [1], [2], [3], [4], [5], [6], [7]
PYTHONTHREADDEBUG
PYTHONUNBUFFERED, [1]
PYTHONUSERBASE, [1], [2], [3]
PYTHONVERBOSE, [1]
PYTHONWARNINGS, [1], [2], [3]
PYTHONY2K, [1], [2], [3]
PYTHON_DOM
SystemRoot
TCL_LIBRARY
TEMP
TERM, [1]
TIX_LIBRARY
TK_LIBRARY
TMP, [1]
TMPDIR, [1]
TZ, [1], [2], [3], [4]
USER
USERNAME
USERPROFILE, [1]
USER_BASE
Wimp$ScrapDir
exec_prefix, [1], [2]
ftp_proxy
http_proxy, [1], [2], [3]
nm_proxy, [1]
prefix, [1], [2], [3]

**environment variables**

deleting
setting
EnvironmentError
EnvironmentVarGuard (class in test.test_support)
ENXIO (in module errno)
eof (shlex.shlex attribute)
EOFError
  (built-in exception)
EOPNOTSUPP (in module errno)
EOVERFLOW (in module errno)
EPERM (in module errno)
f_back (frame attribute)
F_BAVAIL (in module statvfs)
F_BFREE (in module statvfs)
F_BLOCKS (in module statvfs)
F_BSIZE (in module statvfs)
f_builtins (frame attribute)
f_code (frame attribute)
f_exc_traceback (frame attribute)
f_exc_type (frame attribute)
f_exc_value (frame attribute)
F_FAVAIL (in module statvfs)
F_FFREE (in module statvfs)
F_FILES (in module statvfs)
F_FLAG (in module statvfs)
F_FRSIZE (in module statvfs)
f_globals (frame attribute)
f_lasti (frame attribute)
f_lineno (frame attribute)
f_locals (frame attribute)
F_NAMEMAX (in module statvfs)
F_OK (in module os)
f_restricted (frame attribute)
f_trace (frame attribute)
fabs() (in module math)
factorial() (in module math)
fail() (unittest.TestCase method)
failfast (unittest.TestResult attribute)
failureException (unittest.TestCase attribute)
failures (unittest.TestResult attribute)
False, [1], [2]
false
False (Built-in object)
(built-in variable)
family (socket.socket attribute)

flatten() (email.generator.Generator method)
flattening
objects
float
built-in function, [1], [2], [3]
float (built-in class)
float_info (in module sys)
float_repr_style (in module sys)
floating point
literals
number
object, [1], [2]
floating point literal
FloatingPointError, [1]
FloatType (in module types)
(in modules types)
flock() (in module fcntl)

floor division
floor() (in module math), [1]
floordiv() (in module operator)

floordiv (module)
flush() (bz2.BZ2Compressor method)
(formatter.writer method)
(io.BufferedWriter method)
(io.IOBase method)
(logging.Handler method)
(logging.StreamHandler method)
(logging.handlers.BufferingHandler method)
(logging.handlers.MemoryHandler method)
fancy_getopt() (in module distutils.fancy_getopt)
FancyGetopt (class in distutils.fancy_getopt)
FancyURLopener (class in urllib)
fatalError() (xml.sax.handler.ErrorHandler method)
Fault (class in xmlrpclib)
faultCode (xmlrpclib.Fault attribute)
faultString (xmlrpclib.Fault attribute)
fchdir() (in module os)
fchmod() (in module os)
fchown() (in module os)
FCICreate() (in module msilib)

fcntl
    module
fcntl (module)
fcntl() (in module fcntl), [1]
fd() (in module turtle)
fdatasync() (in module os)
fdopen() (in module os)
Feature (class in msilib)
feature_external_ges (in module xml.sax.handler)
feature_external_pes (in module xml.sax.handler)
feature_namespace_prefixes (in module xml.sax.handler)
feature_namespaces (in module xml.sax.handler)
feature_string_interning (in module xml.sax.handler)
feature_validation (in module xml.sax.handler)
feed() (email.parser.FeedParser method)
    (HTMLParser.HTMLParser method)
    (sgmllib.SGMLParser method)
    (xml.etree.ElementTree.XMLParser method)

(mailbox.MH method)
(mailbox.Mailbox method)
(mailbox.Maildir method)
(mmap.mmap method)
(zlib.Compress method)
(zlib.Decompress method)
flush_softspace() (formatter.formatter method)
flushheaders()
(MimeWriter.MimeWriter method)
flushinp() (in module curses)
FlushKey() (in module _winreg)
fm (module)
fma() (decimal.Context method)
    (decimal.Decimal method)
fmod() (in module math)
fmatch (module)
fmatch() (in module fnmatch)
fmatchcase() (in module fnmatch)
focus() (ttk.Treeview method)
Folder (class in mshlib)
Font Manager, IRIS
fontpath() (in module fm)
ofopen()
for
    statement, [1], [2], [3], [4]
FOR_ITER (opcode)
forget() (in module test.test_support)
    (ttk.Notebook method)
fork() (in module os)
    (in module pty)
ForkingMixIn (class in Socket)
ForkingTCPServer (class in SocketServer)
ForkingUDPServer (class in SocketServer)
forkpty() (in module os)
method)
(xml.sax.xmlreader.IncrementalParser
method)
FeedParser (class in email.parser)
fetch() (imaplib.IMAP4 method)
Fetch() (msilib.View method)
fetchall() (sqlite3.Cursor method)
fetchmany() (sqlite3.Cursor method)
fetchone() (sqlite3.Cursor method)
fflags (select.kevent attribute)
field_size_limit() (in module csv)
fieldnames (csv.csvreader attribute)
fields (uuid.UUID attribute)
fifo (class in asynchat)
file

.ini
.pdbrc
.pythonrc.py
built-in function
byte-code, [1], [2]
configuration
copying
debugger configuration
descriptor
large files
mime.types
object, [1], [2], [3], [4]
path configuration
plist
temporary
user configuration
file (pyclbr.Class attribute)
(pyclbr.Function attribute)
file ...

compileall command line option
file control

Form (class in Tix)
format
str
format (memoryview attribute
(struct.Struct attribute)
format() (built-in function)
(in module locale)
(logging.Formatter method)
(logging.Handler method)
(pprint.PrettyPrinter method
(str method)
(string.Formatter method)
format_exc() (in module traceback)
format_exception() (in module
traceback)
format_exception_only() (in
traceback)
format_field() (string.Formatter
method)
format_help()
(argparse.ArgumentParser method)
format_list() (in module traceback)
format_stack() (in module traceback)
format_stack_entry() (bdb.Bdb
method)
format_string() (in module locale)
format_tb() (in module traceback)
format_usage()
(argparse.ArgumentParser method)
formataddr() (in module email
formatargspec() (in module
formatargvalues() (in module
formatdate() (in module email
FormatError
FormatError() (in module ctypes)
formatException() (logging.Formatter
method)
formatmonth()
UNIX

file name
temporary

file object
POSIX
file() (built-in function)
  (posixfile.posixfile method)

file-like object
file_created() (built-in function)
file_dispatcher (class in asyncore)
file_open() (urllib2.FileHandler method)
file_size (zipfile.ZipInfo attribute)
file_wrapper (class in asyncore)
filecmp (module)
fileConfig() (in module logging.config)
FileCookieJar (class in cookielib)
FileEntry (class in Tix)
FileHandler (class in logging)
  (class in urllib2)
FileInput (class in fileinput)
fileinput (module)
FileIO (class in io)
filenono() (in module fileinput)
filename (cookielib.FileCookieJar attribute)
  (doctest.DocTest attribute)
  (zipfile.ZipInfo attribute)
filename() (in module fileinput)
filename_only (in module tabnanny)
filenames
  pathname expansion
  wildcard expansion
fileno() (file method)
  (SocketServer.BaseServer method)
  (hotshot.Profile method)
  (httpplib.HTTPResponse method)
  (in module fileinput)

(calendar.HTMLCalendar me
  (calendar.TextCalendar method)
formarter
  module
Formatter (class in logging)
    (class in string)
formatter (httplib.HTTPResponse attribute)
    (module)
formatTime() (logging.Formatter method)
formatting, string (%)
formatwarning() (in module warnings)
formatyear() (calendar.HTMLCalendar method)
  (calendar.TextCalendar method)
formatyearpage() (calendar.HTMLCalendar me
FORMS Library
forward() (in module turtle)
found_terminator() (asynchat.async_chat method)
fp (rfc822.Message attribute)
fpectl (module)
fpectl (module)
Fraction (class in fractions)
fractions (module)
frame
  execution, [1]
  object
frame (ScrolledText.ScrolledText attribute)
FrameType (in module types)
FrameWork
  module
FrameWork (module)
free
(io.IOBase method)
(multiprocessing.Connection method)
(ossaudiodev.oss_audio_device method)
(ossaudiodev.oss_mixer_device method)
(select.epoll method)
(select.kqueue method)
(socket.socket method)
(telnetlib.Telnet method)
fileopen() (in module posixfile)
FileSelectBox (class in Tix)
FileType (class in argparse)
(in module types), [1]
FileWrapper (class in wsgiref.util)
fill() (in module textwrap)
(in module turtle)
(textwrap.TextWrapper method)
fillcolor() (in module turtle)
filter (2to3 fixer)
Filter (class in logging)
filter (select.kevent attribute)
filter() (built-in function)
(in module curses)
(in module fnmatch)
(in module future_builtins)
(logging.Filter method)
(logging.Handler method)
(logging.Logger method)
(filterwarnings() (in module warnings)
finalization, of objects
finalize_options()
(distutils.cmd.Command method)
finally
  keyword, [1], [2], [3], [4]
find() (doctest.DocTestFinder method)
free()
freeze utility
freeze_form() (fl.form method)
freeze_support() (in module multiprocessing)
frexp() (in module math)
from
  keyword
  statement
from_address() (ctypes._CData method)
from_buffer() (ctypes._CData method)
from_buffer_copy() (ctypes._CData method)
from_decimal() (fractions.Fraction method)
from_float() (decimal.Decimal method)
  (fractions.Fraction method)
from_iterable() (itertools.chain method)
from_param() (ctypes._CData method)
from_splittable() (email.charset.Charset method)
frombuf() (tarfile.TarInfo method)
fromchild (popen2.Popen3 attribute)
fromfd() (in module socket)
select.epoll method)
(select.kqueue method)
fromfile() (array.array method)
fromhex() (float method)
fromkeys() (collections.Counter method)
  (dict method)
fromlist() (array.array method)
fromordinal() (datetime.date method)
(in module gettext)
(in module string)
(mmap.mmap method)
(str method)
(xml.etree.ElementTree.Element method)
(xml.etree.ElementTree.ElementTree method)
find_first() (fl.form method)
find_global() (pickle protocol)
find_last() (fl.form method)
find_library() (in module ctypes.util)
find_library_file()
(distutils.ccompiler.CCompiler method)
find_loader() (in module pkgutil)
find_longest_match()
(difflib.SequenceMatcher method)
find_module
    finder
find_module() (imp.NullImporter method)
    (in module imp)
    (zipimport.zipimporter method)
find_msvcrt() (in module ctypes.util)
find_user_password()
(urllib2.HTTPPasswordMgr method)
findall() (in module re)
    (re.RegexObject method)
    (xml.etree.ElementTree.Element method)
    (xml.etree.ElementTree.ElementTree method)
findCaller() (logging.Logger method)
finder, [1]
    find_module
findertools (module)
findfactor() (in module audioop) method)
    (datetime.datetime class)
fromstring() (array.array method)
    (in module xml.etree.ElementTree)
fromstringlist() (in module xml.etree.ElementTree)
fromtarfile() (tarfile.TarInfo method)
fromtimestamp() (datetime.datetime method)
    (datetime.datetime class)
fromunicode() (array.array method)
fromutcl() (datetime.tzinfo method)
frozenset
    object, [1]
frozenset (built-in class)
fstat() (in module os)
fstatvfs() (in module os)
fsum() (in module math)
fsync() (in module os)
FTP
    ftplib (standard module)
    protocol, [1]
FTP (class in ftplib)
ftp_open() (urllib2.FTPHandler method)
ftp_proxy
FTP_TLS (class in ftplib)
FTPHandler (class in urllib2)
ftplib (module)
ftpmirror.py
ftruncate() (in module os)
Full
full() (multiprocessing.Queue method)
    (Queue.Queue method)
func (functools.partial attribute)
func_closure (function attribute)
func_code (function attribute)
    (function object attribute)
G.722
gaierror
gamma() (in module math)
gammavariate() (in module random)
garbage (in module gc)
garbage collection, [1]
gather() (curses.textpad.Textbox method)
gauss() (in module random)

c (module)
gcd() (in module fractions)
gdbm
    module, [1], [2]
db (module)
ge() (in module operator)
gen_lib_options() (in module distutils.ccompiler)
gen_preprocess_options() (in module distutils.ccompiler)
gen_uuid() (in module msilib)
generate_help() (distutils.fancy_getopt.FancyGetopt method)
generate_tokens() (in module tokenize)
generator, [1]
    expression
    function, [1], [2]
    iterator, [1]
    object, [1], [2]
Generator (class in email.generator)
generator expression, [1]
GeneratorExit
    exception
GeneratorType (in module types)
generic
    special attribute
generic_visit() (ast.NodeVisitor method)
genops() (in module pickletools)

GetErrorString()
geteuid() (in module os)
getEvent()
(xml.dom.pulldom.DOMEventStream method)
getEventCategory()
(logging.har method)
getEventType()
(logging.har method)
getException()
(xml.sax.xmlmethod)
GetFieldCount()
getfile() (in r
getfilesystemencoding()
(getfirst() (c
getfirstmatchingheader()
(rfc822.Message method)
getfloat()
(ConfigParser.RawConfigParser method)
getfmts()
(ossaudiode
getfqdn() (in
getframeinfo
getframerate()
(sunau.A
(wave.W
getfullname
getgid() (in r
gensuitemodule (module)
get() (ConfigParser.ConfigParser method)
    (ConfigParser.RawConfigParser method)
    (Queue.Queue method)
    (dict method)
    (email.message.Message method)
    (in module webbrowser)
    (mailbox.Mailbox method)
    (multiprocessing.Queue method)
    (multiprocessing.multiprocessing.queues.SimpleQueue method)
    (multiprocessing.pool.AsyncResult method)
    (ossaudiodev.oss_mixer_device method)
    (rfc822.Message method)
    (ttk.Combobox method)
    (xml.etree.ElementTree.Element method)
get_all() (email.message.Message method)
    (wsgiref.headers.Headers method)
get_all_breaks() (bdb.Bdb method)
get_app() (wsgiref.simple_server.WSGIServer method)
get_archive_formats() (in module shutil)
get_begidx() (in module readline)
get_body_encoding() (email.charset.Charset method)
get_boundary() (email.message.Message method)
get_break() (bdb.Bdb method)
get_breaks() (bdb.Bdb method)
get_buffer() (xdrlib.Packer method)
    (xdrlib.Unpacker method)
get_ca_certs() (ssl.SSLContext method), [1]
get_channel_binding() (ssl.SSLSocket method)
get_charset() (email.charset.Charset method)
get_charsets() (email.message.Message method)
get_children() (symtable.SymbolTable method)
    (ttk.Treeview method)
get_close_matches() (in module difflib)
get_code() (imputil.BuiltinImporter method)
getgrall() (in module grp)
getgrgid() (in module grp)
getgrnam() (in module grp)
getgroups() (in module os)
getheader() (rfc822.Message method)
gethostbyaddr() (in module socket)
gethostbyname() (in module socket)
gethostbyname_ex() (in module socket)
gethostname() (in module socket)
getincrementaldecoder() (in module codecs)
getincrementalencoder() (in module codecs)
getinfo() (zipfile.ZipFile method)
getinnerframetree() (xml.parsers.expat.xmlparser method)
getint() (codecs.getincrementaldecoder() (in module codecs) method)
GetInteger() (msilib.Record method)
getiterator() (xml.etree.ElementTree method)
getitimer() (in module signal)
getkey() (curses.window method)
getlast() (mhlib.Folder method)
GetLastError() (in module ctypes)
getLength() (xml.sax.xmlreader.Attributes method)
getLevelName() (in module logging)
getline() (in module linecache)
(imutil.Importer method)
(zipimport.zipimporter method)
get_completer() (in module readline)
get_completer_delims() (in module readline)
get_completion_type() (in module readline)
get_config_h_filename() (in module distutils.sysconfig)
    (in module sysconfig)
get_config_var() (in module distutils.sysconfig)
    (in module sysconfig)
get_config_vars() (in module distutils.sysconfig)
    (in module sysconfig)
get_content_charset() (email.message.Message method)
get_content_maintype() (email.message.Message method)
get_content_subtype() (email.message.Message method)
get_content_type() (email.message.Message method)
get_count() (in module gc)
get_current_history_length() (in module readline)
get_data() (in module pkgutil)
    (urllib2.Request method)
    (zipimport.zipimporter method)
get_date() (mailbox.MaildirMessage method)
get_default() (in module gc)
get_default() (argparse.ArgumentParser method)
get_default_compiler() (in module distutils.ccompiler)
get_default_domain() (in module nis)
get_default_type() (email.message.Message method)
get_default_verify_paths() (in module ssl)
get_dialect() (in module csv)
get_directory() (in module fl)
get_docstring() (in module ast)
get_doctest() (doctest.DocTestParser method)
get_endidx() (in module readline)
get_environ()
    (wsgiref.simple_server.WSGIRequestHandler method)
get_errno() (in module ctypes)
get_examples() (doctest.DocTestParser method)
get_field() (string.Formatter method)
get_line_num() (xml.sax.xmlreader.Locator method)
getlist() (cgi.FieldStorage method)
geloadavg() (in module os)
gettext() (in module gettext)
getLogger() (in module logging)
getLoggerClass() (in module logging)
getlogin() (in module os)
getmaintype() (email.message.Message method)
gettext() (in module gettext)
getmaxyx() (curses.window method)
getmcolor() (in module fl)
getmember() (tarfile.TarFile method)
getmembers() (tarfile.TarFile method)
getMessage() (logging.LogRecord method)
    (xml.sax.saxutils method)
getMessageID() (logging.handlers.NTEventLogHandler method)
getmodule() (in module inspect)
getmoduleinfo() (in module inspect)
getmodulename() (in module inspect)
getmouse() (in module curses)
getmro() (in module inspect)
getmtime() (in module os.path)
getname() (threading.Thread method)
getNameByQName()
get_file() (mailbox.Babyl method)
  (mailbox.MH method)
  (mailbox.MMDF method)
  (mailbox.Mailbox method)
  (mailbox.Maildir method)
  (mailbox.maildir method)
get_file_breaks() (bdb.Bdb method)
get_filename() (email.message.Message method)
  (in module fl)
  (zipimport.zipimporter method)
get_flags() (mailbox.MaildirMessage method)
  (mailbox.MMDFMessage method)
  (mailbox.mboxMessage method)
get_folder() (mailbox.Maildir method)
  (mailbox.MH method)
get_frees() (symtable.Function method)
get_from() (mailbox.mboxMessage method)
  (mailbox.MMDFMessage method)
get_full_url() (urllib2.Request method)
get_globals() (symtable.Function method)
get_grouped_opcodes() (difflib.SequenceMatcher method)
get_header() (urllib2.Request method)
get_history_item() (in module readline)
get_history_length() (in module readline)
get_host() (urllib2.Request method)
get_id() (symtable.SymbolTable method)
get_ident() (in module thread)
get_identifiers() (symtable.SymbolTable method)
get_importer() (in module pkgutil)
get_info() (mailbox.MaildirMessage method)
GET_ITER (opcode)
get_labels() (mailbox.Babyl method)
  (mailbox.BabylMessage method)
get_last_error() (in module ctypes)
get_line_buffer() (in module readline)
get_lineno() (symtable.SymbolTable method)
get_loader() (in module pkgutil)
  (xml.sax.xmlmethod)
get_nameinfoc
get_names()
get_names()
get_filenames()
getnode
getnframes()
getnframes()
getnode() (in
getopt (mod
getopt()
getouterframes()
gtoutput() (in
getpagesize()
gtparams()
gtparyx() (in
getpeercert()
gtpeernam
getpen() (in
getpgid() (in
get_locals() (symtable.Function method)
get_logger() (in module multiprocessing)
get_magic() (in module imp)
get_makefile_filename() (in module distutils.sysconfig)
    (in module sysconfig)
get_matching_blocks() (difflib.SequenceMatcher method)
get_message() (mailbox.Mailbox method)
get_method() (urllib2.Request method)
get_methods() (symtable.Class method)
get_mouse() (in module fl)
get_name() (symtable.Symbol method)
    (symtable.SymbolTable method)
get_namespace() (symtable.Symbol method)
get_namespaces() (symtable.Symbol method)
get_nonstandard_attr() (cookielib.Cookie method)
get_nowait() (multiprocessing.Queue method)
    (Queue.Queue method)
get_objects() (in module gc)
get_opcodes() (difflib.SequenceMatcher method)
get_option() (optparse.OptionParser method)
get_option_group() (optparse.OptionParser method)
get_option_order() (distutils.fancy_getopt.FancyGetopt method)
get_origin_req_host() (urllib2.Request method)
get_osfhandle() (in module msvcr)
get_output_charset() (email.charset.Charset method)
get_param() (email.message.Message method)
get_parameters() (symtable.Function method)
get_params() (email.message.Message method)
get_path() (in module sysconfig)
get_path_names() (in module sysconfig)
get_paths() (in module sysconfig)
get_pattern() (in module fl)
get_payload() (email.message.Message method)
get_platform() (in module distutils.util)
    (in module sysconfig)
get_poly() (in module turtle)
get_position() (xdrlib.Unpacker method)
get_visible() (mailbox.BabylMessage method)
getabouttext() (FrameWork.Application method)
getacl() (imaplib.IMAP4 method)
getaddr() (rfc822.Message method)
getaddresses() (in module email.utils)
getaddrinfo() (in module socket)
getaddrlist() (rfc822.Message method)
getallmatchingheaders() (rfc822.Message method)
getannotation() (imaplib.IMAP4 method)
getargspec() (in module inspect)
GetArgv() (in module EasyDialogs)
getargvalues() (in module inspect)
getatime() (in module os.path)
getattr() (built-in function)
getAttribute() (xml.dom.Element method)
getAttributeNode() (xml.dom.Element method)
getAttributeNodeNS() (xml.dom.Element method)
getAttributeNS() (xml.dom.Element method)
GetBase() (xml.parsers.expat.xmlparser method)
getbegyx() (curses.window method)
getbkgd() (curses.window method)
getboolean() (ConfigParser.RawConfigParser method)
getByteStream() (xml.sax.xmlreader.InputSource method)
getcallargs() (in module inspect)
getcanvas() (in module turtle)
getcaps() (in module mailcap)
getch() (curses.window method)
getCharacterStream() (xml.sax.xmlreader.InputSource method)
getche() (in module msvcrt)
getcheckinterval() (in module sys)
getChild() (logging.Logger method)
getChildNodes() (compiler.ast.Node method)
getChildren() (compiler.ast.Node method)
getchildren() (xml.etree.ElementTree.Element method)
getclasstree() (in module inspect)
GetColor() (in module ColorPicker)
GetColumnInfo() (msilib.View method)
getColumnNumber() (xml.sax.xmlreader.Locator method)
getcomments() (in module inspect)
getcompname() (aifc.aifc method)
    (sunau.AU_read method)
    (wave.Wave_read method)
getcomptype() (aifc.aifc method)
    (sunau.AU_read method)
    (wave.Wave_read method)
getContentHandler() (xml.sax.xmlreader.XMLReader method)
getcontext() (in module decimal)
    (mhlh.MH method)
GetCreatorAndType() (in module MacOS)
getctime() (in module os.path)
getcurrent() (mhlh.Folder method)
getcwd() (in module os)
getcwdu (2to3 fixer)
getcwdu() (in module os)
getdate() (rfc822.Message method)
getdate_tz() (rfc822.Message method)
getdecoder() (in module codecs)
getdefaultencoding() (in module sys)
getdefaultlocale() (in module locale)
getdefaulttimeout() (in module socket)
getdlopenflags() (in module sys)
getdoc() (in module inspect)
getDOMImplementation() (in module xml.dom)
getDTDHandler() (xml.sax.xmlreader.XMLReader method)
getEffectiveLevel() (logging.Logger method)
getegid() (in module os)
getElementsByTagName() (xml.dom<Document method)
    (xml.dom.Element method)
getElementsByTagNameNS() (xml.dom.Document method)
    (xml.dom.Element method)
getencoder() (in module codecs)
getencoding() (mimetypes.Message method)
getEncoding() (xml.sax.xmlreader.InputSource method)
geturl() (urlparse.ParseResult method)
getuser() (in module getpass)
getuserbase
getusersitepackages() (in module site)
getvalue() (io.StringIO method)
    (StringIO.StringIO method)
getWeakrefs() (xml.sax.xmlreader method)
getWelcome()
getwelcome
geturl() (urlparse.ParseResult method)
getuser() (in module getpass)
getuserbase
getusersitepackages() (in module site)
getvalue() (io.StringIO method)
    (StringIO.StringIO method)
getWeakrefs() (xml.sax.xmlreader method)
getWelcome()
getwelcome
getwch() (in module msvcrt)
getwche() (in module msvcrt)
getWeakrefs() (xml.sax.xmlreader method)
getWelcome()
getwelcome
getwch() (in module msvcrt)
getwche() (in module msvcrt)
getWeakrefs() (xml.sax.xmlreader method)
getWelcome()
getwelcome
getwelcome
GIL, [1]
GL (module)
glob
module
glob (module)
glob() (in module msilib.Directory method)
glob() (in module msilib.Directory method)
globe
name bind
globe (module)
glob() (in module msilib.Directory method)
glob() (in module msilib.Directory method)
globe
name bind
namespace
globe (module)
globe (module)
getEntityResolver() (xml.sax.xmlreader.XMLReader method)
getenv() (in module os)
getErrorHandler() (xml.sax.xmlreader.XMLReader method)
globals
    built-in function
globals() (built-in function)
globs (doctest.DocTest attribute)
gmtime() (in module time)
gname (tarfile.TarInfo attribute)
GNU_FORMAT (in module tarfile)
gnu_getopt() (in module getopt)
got (doctest.DocTestFailure attribute)
goto() (in module turtle)
gravity
Graphical User Interface
GREATER (in module token)
GREATEREQUAL (in module token)
Greenwich Mean Time
grey22grey() (in module imageop)
grey2grey2() (in module imageop)
grey2grey4() (in module imageop)
grey2mono() (in module imageop)
grey42grey() (in module imageop)
group() (nntplib.NNTP method)
groupby() (in module itertools)
groupdict() (re.MatchObject method)
groupindex (re.RegexObject attribute)
groups (re.RegexObject attribute)
groups() (re.RegexObject attribute)
grp (module functools)
gt() (in module operator)
guess_all_extensions() (in module mimetypes)
guess_extension() (in module mimetypes)
guess_scheme() (in module wsgiref.util)
guess_type()
   (mimetypes)
GUI
gzip (module)
GzipFile (class)
halfdelay() (in module curses)
handle an exception
handle()
(BaseHTTPServer.BaseHTTPRequestHandler method)
  (SocketServer.BaseRequestHandler method)
  (logging.Handler method)
  (logging.Logger method)
  (logging.NullHandler method)
  (wsgiref.simple_server.WSGIRequestHandler method)
handle_accept() (asyncore.dispatcher method)
handle_charref() (HTMLParser.HTMLParser method)
  (sgmlib.SGMLParser method)
handle_close() (asyncore.dispatcher method)
handle_comment() (HTMLParser.HTMLParser method)
  (sgmlib.SGMLParser method)
handle_connect() (asyncore.dispatcher method)
handle_data() (HTMLParser.HTMLParser method)
  (sgmlib.SGMLParser method)
handle_decl() (HTMLParser.HTMLParser method)
  (sgmlib.SGMLParser method)
handle_endtag() (HTMLParser.HTMLParser method)
  (sgmlib.SGMLParser method)
handle_entityref() (HTMLParser.HTMLParser method)
  (sgmlib.SGMLParser method)
handle_error() (asyncore.dispatcher method)
  (SocketServer.BaseServer method)
handle_expt() (asyncore.dispatcher method)
handle_image() (htmllib.HTMLParser method)
handle_one_request()
(BaseHTTPServer.BaseHTTPRequestHandler method)
handle_pi() (HTMLParser.HTMLParser method)
handle_read() (asyncore.dispatcher method)
handle_request()
(SimpleXMLRPCServer.CGIXMLRPCRequestHandler method)
   (SocketServer.BaseServer method)
handle_startendtag() (HTMLParser.HTMLParser method)
   (sgmllib.SGMLParser method)
handle_timeout() (SocketServer.BaseServer method)
handle_write() (asyncore.dispatcher method)
handle_Error() (logging.Handler method)
   (logging.handlers.SocketHandler method)
handler
   exception
handler() (in module cgitb)
HAS_ALPN (in module ssl)
has_children() (symtable.SymbolTable method)
has_colors() (in module curses)
has_data() (urllib2.Request method)
HAS_ECDH (in module ssl)
has_exec() (symtable.SymbolTable method)
has_extn() (smtplib.SMTP method)
has_function() (distutils.ccompiler.CCompiler method)
has_header() (csv.Sniffer method)
   (urllib2.Request method)
has_ic() (in module curses)
has_il() (in module curses)
has_import_star() (symtable.SymbolTable method)
has_ipv6 (in module socket)
has_key (2to3 fixer)
has_key() (bsddb.bsddbobject method)
   (dict method)
   (email.message.Message method)
   (in module curses)
   (mailbox.Mailbox method)

_HKEY_LOCAL_MACHINE (in module _winreg)
_HKEY_PERFORMANCE_DATA (in module _winreg)
_HKEY_USERS (in module _winreg)
HKEY_USER (in module _winreg)
hline() (curses.window method)
HList (class in Tix)
hls_to_rgb() (in module colorsys)
hmac (module)
HOME, [1], [2], [3]
home() (in module os)
HOMEDRIVE, [1]
HOMEPATH, [1]
hook_compressed() (in module fileinput)
hook_encoded() (in module netrc)
hotshot (module)
hotshot.stats (module)
hour (datetime.datetime method)
   (datetime.time method)
HRESULT (class in ctypes)
hStdError (subprocess.STARTUPINFO method)
hStdInput (subprocess.STARTUPINFO method)
hStdOutput (subprocess.STARTUPINFO method)
hsv_to_rgb() (in module colorsys)
ht() (in module turtle)
HTML, [1], [2]
HTMLCalendar (in module calendar)
HtmlDiff (class in difflib)
HtmlDiff.__init__
HtmlDiff.make_file
has_nonstandard_attr() (cookielib.Cookie method)
HAS_NPN (in module ssl)
has_option() (ConfigParser.RawConfigParser
method)

(optparse.OptionParser method)
has_section() (ConfigParser.RawConfigParser
method)
HAS_SNI (in module ssl)
hasattr() (built-in function)
hasAttribute() (xml.dom.Element method)
hasAttributeNS() (xml.dom.Element method)
hasAttributes() (xml.dom.Node method)
hasChildNodes() (xml.dom.Node method)
hascompare (in module dis)
hasconst (in module dis)
hasFeature() (xml.dom.DOMImplementation method)
hasfree (in module dis)

hash
   built-in function, [1], [2]
hash character
hash() (built-in function)
hash.block_size (in module hashlib)
hash.digest_size (in module hashlib)
hashable, [1]
Hashable (class in collections)
hashlib (module)
hashlib.algorithms (in module hashlib)
hashopen() (in module bsddb)
hasjabs (in module dis)
hasjrel (in module dis)
haslocal (in module dis)
hasname (in module dis)
HAVE_ARGUMENT (opcode)
have_unicode (in module test.test_support)
head() (nntplib.NNTP method)
Header (class in email.header)
header_encode() (email.charset.Charset method)
header_encoding (email.charset.Charset attribute)

HtmlDiff.make_table() (difflib)
htmlentitydefs (in
htmllib
module)
htmllib (module)
HTMLParseError
HTMLParser (class
(module)
htonl() (in module
htons() (in modu
HTTP
   httpbin (standard
   protocol, [1],
http_error_301() (urllib2.HTTPRe
method)
http_error_302() (urllib2.HTTPRe
method)
http_error_303() (urllib2.HTTPRe
method)
http_error_307() (urllib2.HTTPRe
method)
http_error_401() (urllib2.HTTPBa
method)
(http_error_407() (urllib2.ProxyBa
method)
(http_error_407() (urllib2.ProxyBa
method)
header_items() (urllib2.Request method)
header_offset (zipfile.ZipInfo attribute)
HeaderError
HeaderParseError
headers
    MIME, [1]
headers
(BaseHTTPServer.BaseHTTPRequestHandler attribute)
Headers (class in wsgiref.headers)
headers (rfc822.Message attribute)
    (xmlrpclib.ProtocolError attribute)
heading() (in module turtle)
    (ttk.Treeview method)
heapify() (in module heapq)
heapmin() (in module msvcrt)
heappop() (in module heapq)
heappush() (in module heapq)
heappushpop() (in module heapq)
heapq (module)
heappopreplace() (in module heapq)
heho() (smtplib.SMTP method)
help
    built-in function
        online
help (optparse.Option attribute)
help() (built-in function)
    (nntplib.NNTP method)
herror
hex
    built-in function
http_error_auth_ (urllib2.AbstractE
    method)
    (urllib2.AbstractE
    method)
http_error_defau (urllib2.BaseHan
http_error_nnn() (method)
http_open() (urllib
HTTP_PORT (in
http_proxy, [1], [:
http_response() (urllib2.HTTPErr
http_version (wsgiref.handler:
HTTPBasicAuthI
urllib2)
HTTPConnectionio
HTTPCookiePro
urllib2)
httpd
HTTPDefaultErr(urllib2)
HTTPDigestAut\urllib2)
HTTPError
HTTPErrorProce
HTTPException
HTTPHandler (c
logging.handlers
    (class in urlli
httplib (module)
HTTPMessage ( HTTPPassw
HTTPPasswordM
http_error_auth_ (urllib2.AbstractE
    method)
    (urllib2.AbstractE
    method)
http_error_defau (urllib2.BaseHan
http_error_nnn() (method)
http_open() (urllib
HTTP_PORT (in
http_proxy, [1], [:
http_response() (urllib2.HTTPErr
http_version (wsgiref.handler:
HTTPBasicAuthI
urllib2)
HTTPConnectionio
HTTPCookiePro
urllib2)
httpd
HTTPDefaultErr(urllib2)
HTTPDigestAut\urllib2)
HTTPError
HTTPErrorProce
HTTPException
HTTPHandler (c
logging.handlers
    (class in urlli
httplib (module)
HTTPMessage ( HTTPPassw
HTTPPasswordM
HTTPRedirectHandler (class in urllib2)

HTTPResponse

https_open() (url method)

HTTPS_PORT (in module httplib)

https_response()

(available as urllib2.HTTPErrorProcessor method)

HTTPSConnection (class in httplib)

HTTPServer (class in BaseHTTPServer)

HTTPSHandler (class in urllib2)

hypertext

hypot() (in module math)
I (in module re)
I/O control
  POSIX
  UNIX
    buffering, [1], [2]
    tty
iadd() (in module operator)
iand() (in module operator)
IC (class in ic)
ic (module)
icglue
  module
iconcat() (in module operator)
icopen (module)
id
  built-in function
id() (built-in function)
    (unittest.TestCase method)
idcok() (curses.window method)
ident (in module cd)
    (select.kevent attribute)
    (threading.Thread attribute)
identchars (cmd.Cmd attribute)
identifier, [1]
identify() (ttk.Notebook method)
    (ttk.Treeview method)
    (ttk.Widget method)
identify_column() (ttk.Treeview method)
identify_element() (ttk.Treeview method)
identify_region() (ttk.Treeview method)
identify_row() (ttk.Treeview method)
identity
  test
  identity of an object
internalSubset (xml.dom attribute)
Internet
Internet Config
interpolation, string (%)
InterpolationDepthError
InterpolationError
InterpolationMissingOptionError
InterpolationSyntaxError
interpreted
interpreter
interpreter lock
interpreter prompts
interrupt() (sqlite3.Connection method)
interrupt_main() (in module threading.Thread)
intersection() (set method)
intersection_update() (set method)
IntlText (class in aetypes)
IntlWritingCode (class in aetypes)
intro (cmd.Cmd attribute)
IntType (in module types)
    (in modules types)
InuseAttributeErr
inv() (in module operator)
InvalidAccessErr
InvalidCharacterErr
InvalidModificationErr
InvalidOperation (class in decimal)
InvalidStateErr
InvalidURL
inversion
invert() (in module operator)
invocation
io (module)
IOBase (class in io)
idioms (2to3 fixer)
idi() (in module operator)
IDLE, [1]
idle() (FrameWork.Application method)
IDLESTARTUP
idlok() (curses.window method)
IEEE-754
if
    statement, [1]
ifilter() (in module itertools)
ifilterfalse() (in module itertools)
ifloordiv() (in module operator)
iglob() (in module glob)
ignoreWhitespace()
(xml.sax.handler.ContentHandler method)
ignore_errors() (in module codecs)
IGNORE_EXCEPTION_DETAIL (in module doctest)
ignore_patterns() (in module shutil)
IGNORECASE (in module re)
ihave() (nntplib.NNTP method)

ihooks
    module
    ilshift() (in module operator)
im_class (method attribute), [1]
im_func (method attribute), [1], [2], [3]
im_self (method attribute), [1], [2]
imag (numbers.Complex attribute)
imageop (module)
imaginary literal
imap() (in module itertools)
    (multiprocessing.pool.multiprocessing.Pool
        method)
IMAP4
    protocol
IMAP4 (class in imaplib)
IMAP4.abort
IMAP4.error
ioctl() (in module fcntl)
    (socket.socket method)
IOError
ior() (in module operator)
irop() (in module operator)
irepeat() (in module operator)
IRIS Font Manager
IRIX
    threads
irshift() (in module operator)
is
    operator, [1]
is not
    operator, [1]
is_() (in module operator)
is_alive() (multiprocessing method)
    (threading.Thread method)
is_assigned() (symtable.
    method)
is_blocked()
    (cookielib.DefaultCookie
    is_buitlin() (in module im
    is_canonical() (decimal.
        (decimal.Decimal method)
    IS_CHARACTER_JUNK
    difflib)
is_data() (multifile.MultiF
    is_declared_global() (symtable.
    method)
is_empty() (asynchat.fifo
    is_expired() (cookielib.C
    is_finite() (decimal.Conte
        (decimal.Decimal method)
    is_free() (symtable.Symbol
    is_frozen() (in module im
    is_global() (symtable.Symbol
    is_hop_by_hop() (in mod
IMAP4.readonly
IMAP4_SSL
    protocol
IMAP4_SSL (class in imaplib)
IMAP4_stream
    protocol
IMAP4_stream (class in imaplib)
imap_unordered()
    (multiprocessing.pool.multiprocessing.Pool method)
imaplib (module)
imgfile (module)
imghdr (module)
immedok() (curses.window method)
imutable
    data type
    object, [1], [2]
immutable object
immutable sequence
    object
immutable types
    subclassing
ImmutableSet (class in sets)
imod() (in module operator)
imp
    module
imp (module)
ImpImporter (class in pkgutil)
ImpLoader (class in pkgutil)
import
    statement, [1], [2], [3], [4]
import (2to3 fixer)
import_file() (imputil.DynLoadSuffixImporter method)
import_fresh_module() (in module test.test_support)
IMPORT_FROM (opcode)
import_module() (in module importlib)
is_imported() (symtable.method)
is_infinite() (decimal.Context method)
is_integer() (float method)
is_jython (in module test)
IS_LINE_JUNK() (in module)
is_linetouched() (curses.window method)
is_local() (symtable.Symbol method)
is_multipart() (email.message.Message method)
is_namespace() (symtable.Symbol method)
is_nan() (decimal.Context method)
is_not() (in module operator)
is_not_allowed()
(cookielib.DefaultCookiePolicy method)
is_optimized() (symtable.Symbol method)
is_package() (zipimport.zipimporter method)
is_parameter() (symtable.Symbol method)
is_python_build() (in module sysconfig)
is_qnan() (decimal.Decimal method)
is_referenced() (symtable.Symbol method)
is_resource_enabled() (in module test.test_support)
is_scriptable() (in module gensuitemodule)
inclusive
  or
Incomplete
IncompleteRead
incr_item(), [1]
increment_lineno() (in module ast)
IncrementalDecoder (class in codecs)
IncrementalEncoder (class in codecs)
IncrementalNewlineDecoder (class in io)
IncrementalParser (class in xml.sax.xmlreader)
indent (doctest.Example attribute)
INDENT (in module token)
INDEX token
indentation
IndentationError
Independent JPEG Group
index (in module cd)
index operation
index() (array.array method)
  (in module operator)
  (in module string)
  (list method)
  (str method)
  (ttk.Notebook method)
  (ttk.Treeview method)
IndexError
indexOf() (in module operator)
IndexSizeErr
indices() (slice method)
inet_aton() (in module socket)
inet_ntoa() (in module socket)
inet_ntop() (in module socket)
inet_pton() (in module socket)
Inexact (class in decimal)
infile (shlex.shlex attribute)
iscode() (in module inspect)
iscomment() (rfc822.Message)
isctrl() (in module curses)
isDaemon() (threading.Thread)
isdatadescriptor() (in module codecs)
isdecimal() (unicode method)
isdev() (tarfile.TarInfo method)
isdigit() (in module curses)
  (str method)
isdir() (in module os.path)
  (tarfile.TarInfo method)
isdisjoint() (set method)
isdown() (in module turtle)
iselement() (in module xml.etree.ElementTree)
isenabled() (in module gc)
isEnabledFor() (logging.Logger)
isendwin() (in module curses)
ISEOF() (in module token)
isspace() (in module parser)
  (parser.ST method)
isfifo() (tarfile.TarInfo)
isfile() (in module os.path)
  (tarfile.TarInfo method)
isfirstline() (in module fileinput)
isframe() (in module inspect)
isfunction() (in module inspect)
isgenerator() (in module inspect)
isgeneratorfunction() (in module inspect)
isgetsetdescriptor() (in module inspect)
isgraph() (in module cstring)
isheader() (rfc822.Message)
isinf() (in module cmath)
  (in module math)
isinstance (2to3 fixer)
isinstance() (built-in function)
iskeyword() (in module keyword)
islast() (rfc822.Message)
Infinity, [1]
info() (gettext.NullTranslations method)
   (in module logging)
   (logging.Logger method)
infolist() (zipfile.ZipFile method)
InfoScrap() (in module Carbon.Scrap)
inheretance
init() (in module fm)
   (in module mimetypes)
   init_builtin() (in module imp)
init_color() (in module curses)
in_database() (in module msilib)
in_frozen() (in module imp)
in_pair() (in module curses)
inited (in module mimetypes)
initgroups() (in module os)
initial_indent (textwrap.TextWrapper attribute)
initialize_options() (distutils.cmd.Command method)
initscr() (in module curses)
INPLACE_ADD (opcode)
INPLACE_AND (opcode)
INPLACE_DIVIDE (opcode)
INPLACE_FLOOR_DIVIDE (opcode)
INPLACE_LSHIFT (opcode)
INPLACE_MODULO (opcode)
INPLACE_MULTIPY (opcode)
INPLACE_OR (opcode)
INPLACE_POWER (opcode)
INPLACE_RSHIFT (opcode)
INPLACE_SUBTRACT (opcode)
INPLACE_TRUE_DIVIDE (opcode)
INPLACE_XOR (opcode)
input
   built-in function, [1]
   raw
input (2to3 fixer)
isleap() (in module calendar)
islice() (in module itertools)
islink() (in module os.path)
islink() (tarfile.TarInfo method)
islower() (in module curses)
   (str method)
isMappingType() (in module operator)
ismemberdescriptor() (in inspect)
ismeta() (in module curses)
ismethod() (in module inspect)
ismethoddescriptor() (in inspect)
ismodule() (in module inspect)
ismount() (in module os)
isnan() (in module math)
   (in module math)
ISNONTERMINAL() (in module token)
isNumberType() (in module types)
isnumeric() (unicode method)
isocalendar() (datetime.date method)
(isoformat() (datetime.datetime method)
isoweekday() (datetime.date method)
isprint() (in module curses)
ispunct() (in module curses)
isqueued() (in module fl)
isreadable() (in module pprint)
   (pprint.PrettyPrinter method)
isrecursive() (in module pprint)
   (pprint.PrettyPrinter method)
isreg() (tarfile.TarInfo method)
isReservedKey() (Cookie.Morsel method)
object, [1], [2], [3], [4]
instance() (in module new)
instancemethod() (in module new)
InstanceType (in module types)
instate() (ttk.Widget method)
instr() (curses.window method)
istream (shlex.shlex attribute)
int
    built-in function, [1], [2]
int (built-in class)
    (uuid.UUID attribute)
Int2AP() (in module imaplib)
integer
    division
    division, long
    literals
    literals, long
    object, [1], [2]
    representation
    types, operations on
    integer division
    integer literal
    Integral (class in numbers)
    Integrated Development Environment
    Intel/DVI ADPCM
interact() (code.InteractiveConsole method)
    (in module code)
    (telnetlib.Telnet method)
interactive
    interactive mode
    InteractiveConsole (class in code)
    InteractiveInterpreter (class in code)
    intern (2to3 fixer)
    intern() (built-in function)
    internal type
    internal_attr (zipfile.ZipInfo attribute)
    Internaldate2tuple() (in module imaplib)
method
    itemsize (array.array attribute)
    ItemsView (class in collections)
inter() (built-in function)
    (xml.etree.ElementTree method)
    (xml.etree.ElementTree method)
iter_child_nodes() (in module ast)
iter_fields() (in module ast)
iter_importers() (in module pkgutil)
iter_modules() (in module pkgutil)
iterable
    Iterable (class in collections)
    IterableUserDict (class in collections)
    iterator
    Iterator (class in collections)
    iterator protocol
    iterdecode() (in module codecs)
    iterdump (sqlite3.Connection attribute)
    iterencode() (in module codecs)
    iterfind() (xml.etree.ElementTree method)
    (xml.etree.ElementTree method)
    iteritems() (dict method)
    (mailbox.Mailbox method)
    iterkeyrefs() (weakref.WeakKeyDictionary method)
    iterkeys() (dict method)
    (mailbox.Mailbox method)
    itermonthdates() (calendar method)
    itermonthdays() (calendar method)
method)
itermonthdays2() (calendar method)
iterparse() (in module xml.etree.ElementTree)
itertext() (xml.etree.ElementTree method)
itertools (2to3 fixer)
  (module)
itertools_imports (2to3 fixer)
itervaluerefs() (weakref.WeakValueDictionary method)
itervalues() (dict method)
  (mailbox.Mailbox method)
iterweekdays() (calendar method)
ITIMER_PROF (in module signal)
ITIMER_REAL (in module signal)
ITIMER_VIRTUAL (in module signal)
ItimerError
itruediv() (in module operator)
ixor() (in module operator)
izip() (in module itertools)
izip_longest() (in module itertools)
Jansen, Jack

**Java**

- Java
- language
- java_ver() (in module platform)
- JFIF, [1]
- join() (in module os.path)
  - (Queue.Queue method)
  - (in module string)
  - (multiprocessing.JoinableQueue method)
  - (multiprocessing.Process method)
  - (multiprocessing.pool.multiprocessing.Pool method)
  - (str method)
  - (threading.Thread method)
- join_thread() (multiprocessing.Queue method)
- JoinableQueue (class in multiprocessing)
- joinfields() (in module string)
- jpeg (module)
- js_output() (Cookie.BaseCookie method)
  - (Cookie.Morsel method)

**json**

- json module
- json (module)
- JSONDecoder (class in json)
- JSONEncoder (class in json)
- JUMP_ABSOLUTE (opcode)
- JUMP_FORWARD (opcode)
- JUMP_IF_FALSE_OR_POP (opcode)
- JUMP_IF_TRUE_OR_POP (opcode)
- jumpahead() (in module random)
K

kbhit() (in module msvcrt)
KDEDIR
kevent() (in module select)
key
    (Cookie.Morsel attribute)

**key function**
key/datum pair
KEY_ALL_ACCESS (in module _winreg)
KEY_CREATE_LINK (in module _winreg)
KEY_CREATE_SUB_KEY (in module _winreg)
KEY_ENUMERATE_SUB_KEYS (in module _winreg)
KEY_EXECUTE (in module _winreg)
KEY_NOTIFY (in module _winreg)
KEY_QUERY_VALUE (in module _winreg)
KEY_READ (in module _winreg)
KEY_SET_VALUE (in module _winreg)
KEY_WOW64_32KEY (in module _winreg)
KEY_WOW64_64KEY (in module _winreg)
KEY_WRITE (in module _winreg)
KeyboardInterrupt
    (built-in exception), [1]

KeyPressed
    (in module msvcrt)
KDEDIR
kevent() (in module select)
key
    (Cookie.Morsel attribute)

**key function**
key/datum pair
KEY_ALL_ACCESS (in module _winreg)
KEY_CREATE_LINK (in module _winreg)
KEY_CREATE_SUB_KEY (in module _winreg)
KEY_ENUMERATE_SUB_KEYS (in module _winreg)
KEY_EXECUTE (in module _winreg)
KEY_NOTIFY (in module _winreg)
KEY_QUERY_VALUE (in module _winreg)
KEY_READ (in module _winreg)
KEY_SET_VALUE (in module _winreg)
KEY_WOW64_32KEY (in module _winreg)
KEY_WOW64_64KEY (in module _winreg)
KEY_WRITE (in module _winreg)
KeyboardInterrupt
    (built-in exception), [1]

KeyError
keyname() (in module curses)
keypad() (curses.window method)
keyrefs() (weakref.WeakKeyDictionary method)
keys() (bsddb.bsddbobject method)
    (dict method)
    (email.message.Message method)
    (mailbox.Mailbox method)
    (sqlite3.Row method)
    (xml.etree.ElementTree.Element method)
keys subst() (in module aetools)
KeysView (class in collections)

**keyword**
elif
else, [1], [2], [3], [4]
except
finally, [1], [2], [3], [4]
from
in
yield

Keyword (class in aetypes)
keyword (module)

**keyword argument**
keywords (functools.partial attribute)
k ill() (in module os)
    (subprocess.Popen method)
k illchar() (in module curses)
k illpg() (in module os)
knee
module, [1]
knownfiles (in module mimetypes)
kqueue() (in module select)
kwlist (in module keyword)
L (in module re)
label() (EasyDialogs.ProgressBar method)
LabelEntry (class in Tix)
LabelFrame (class in Tix)
lambda
    expression, [1]
LambdaType (in module types)
LANG, [1], [2], [3], [4]
LANGUAGE, [1]
language
    C, [1], [2], [3], [4]
        Java
        Pascal
large files
LargeZipFile
last (multifile.MultiFile attribute)
last() (bsddb.bsddbobject method)
    (dbhash.dbhash method)
    (nntplib.NNTP method)
last_accepted
    (multiprocessing.connection.Listener attribute)
last_traceback (in module sys), [1]
last_type (in module sys)
last_value (in module sys)
lastChild (xml.dom.Node attribute)
lastcmd (cmd.Cmd attribute)
lastgroup (re.MatchObject attribute)
lastindex (re.MatchObject attribute)
lastpart() (MimeWriter.MimeWriter method)
lastrowid (sqlite3.Cursor attribute)
launch() (in module findertools)
launchurl() (ic.IC method)
    (in module ic)

literals
    complex number
    floating point
    hexadecimal
    integer
    long integer
    numeric
    octal
LittleEndianStructure (class in ctypes)
ljust() (in module string)
    (str method)
LK_LOCK (in module msvcrt)
LK_NBLCK (in module msvcrt)
LK_NBRLCK (in module msvcrt)
LK_RLCK (in module msvcrt)
LK_UNLCK (in module msvcrt)
LMTP (class in smtplib)
ln() (decimal.Context method)
    (decimal.Decimal method)
LNAME
lngettext() (gettext.GNUTranslations method)
    (gettext.NullTranslations method)
    (in module gettext)
load() (Cookie.BaseCookie method)
    (cookielib.FileCookieJar method)
    (in module hotshot.stats)
    (in module json)
    (in module marshal)
    (in module pickle)
    (pickle.Unpickler method)
LOAD_ATTR (opcode)
load_cert_chain() (ssl.SSLContext method)
LOAD_CLOSURE (opcode)
layout() (ttk.Style method)
LBRAVE (in module token)
**LBRL**
LC_ALL, [1]  
(in module locale)
LC_COLLATE (in module locale)
LC_CTYPE (in module locale)
LC_MESSAGES, [1]  
(in module locale)
LC_MONETARY (in module locale)
LC_NUMERIC (in module locale)
LC_TIME (in module locale)
lchflags() (in module os)
lchmod() (in module os)
lchown() (in module os)
LDCXXSHARED
ldexp() (in module math)
LDFLAGS
ldgettext() (in module gettext)
ldngettext() (in module gettext)
le() (in module operator)
leading whitespace
leapdays() (in module calendar)
leaveok() (curses.window method)
left (filecmp.dircmp attribute)
left() (in module turtle)
left_list (filecmp.dircmp attribute)
left_only (filecmp.dircmp attribute)
LEFTSHIFT (in module token)
LEFTSHIFTEQUAL (in module token)
len
   built-in function, [1], [2], [3], [4], [5], [6],
   [7], [8], [9], [10], [11]
len() (built-in function)
length (xml.dom.NamedNodeMap attribute)
   (xml.dom.NodeList attribute)
LESS (in module token)
LESSEQUAL (in module token)
load_compiled() (in module)
LOAD_CONST (opcode)
load_default_certs() (ssl.SSLContext method)
LOAD_DEREF (opcode)
load_dh_params() (ssl.SSLContext method)
load_extension() (sqlite3.Connection method)
LOAD_FAST (opcode)
LOAD_GLOBAL (opcode)
load_global() (pickle protocol)
LOAD_LOCALS (opcode)
load_module
   loader
load_module() (in module
   zipimport.zipimporter
LOAD_NAME (opcode)
load_source() (in module
load_verify_locations() (ssl.SSLContext method)
loader, [1]
   load_module
LoadError
LoadKey() (in module _winreg)
LoadLibrary() (ctypes.LibraryLoader method)
loads() (in module json)
   (in module marshal)
   (in module pickle)
   (in module xmlrpclib)
loadTestsFromModule() (unittest.TestLoader method)
loadTestsFromName() (unittest.TestLoader method)
loadTestsFromNames() (unittest.TestLoader method)
local (class in threading)
localcontext() (in module importlib)
letters (in module string)
level (multifile.MultiFile attribute)
lexical analysis
lexical definitions
lexists() (in module os.path)
lgamma() (in module math)
gettext() (gettext.GNUTranslations method)
   (gettext.NullTranslations method)
   (in module gettext)
lib2to3 (module)
libc_ver() (in module platform)
library (in module dbm)
   (ssl.SSLError attribute)
library_dir_option()
   (distutils.ccompiler.CCompiler method)
library_filename()
   (distutils.ccompiler.CCompiler method)
library_option()
   (distutils.ccompiler.CCompiler method)
LibraryLoader (class in ctypes)
license (built-in variable)
LifoQueue (class in Queue)
light-weight processes
limit_denominator() (fractions.Fraction method)
lin2adpcm() (in module audioop)
lin2alaw() (in module audioop)
lin2lin() (in module audioop)
lin2ulaw() (in module audioop)
line continuation
line joining, [1]
line structure
line() (msilib.Dialog method)
line-buffered I/O
line_buffering (io.TextIOWrapper attribute)
line_num (csv.csvreader attribute)
linenode (module)
lineno (ast.AST attribute)
LOCALE (in module re)
locale (module)
localeconv() (in module locale)
LocaleHTMLCalendar (class in calendar)
LocaleTextCalendar (class in calendar)
localName (xml.dom.Node attribute)
locals
   built-in function
locals() (built-in function)
localtime() (in module time)
Locator (class in xml.sax)
Lock (class in multiprocessing)
Lock() (in module threading)
lock() (mailbox.Babyl method)
   (mailbox.MMDF method)
   (mailbox.Mailbox method)
   (mailbox.Maildir method)
   (mailbox.mbox method)
Lock()
   (multiprocessing.managers.SyncManager method)
lock() (mutex.mutex method)
   (posixfileposixfile method)
lock, interpreter
lock_held() (in module imp)
locked() (thread.lock method)
lockf() (in module fcntl)
locking() (in module msvcrt)
LockType (in module thread)
log() (decimal.Context method)
   (decimal.Decimal method)
(doctest.DocTest attribute)
(doctest.Example attribute)
(pyclbr.Class attribute)
(pyclbr.Function attribute)
(shlex.shlex attribute)
(xml.parsers.expat.ExpatError attribute)
lineno() (in module fileinput)
LINES, [1], [2]
linesep (in module os)
lineterminator (csv.Dialect attribute)
link() (distutils.ccompiler.CCompiler method)
    (in module os)
link_executable()
    (distutils.ccompiler.CCompiler method)
link_shared_lib()
    (distutils.ccompiler.CCompiler method)
link_shared_object()
    (distutils.ccompiler.CCompiler method)
linkmodel (in module MacOS)
linkname (tarfile.TarInfo attribute)
linux_distribution() (in module platform)
list
    assignment, target
    comprehensions, [1]
deletion target
display
empty
expression, [1], [2]
oobject, [1], [2], [3], [4], [5], [6], [7], [8]
target, [1]
type, operations on
list (built-in class)
list comprehension
list() (imaplib.IMAP4 method)
    (multiprocessing.managers.SyncManager method)
    (in module cmath)
log1p() (in module math)
log_date_time_string()
    (BaseHTTPServer.BaseHTTPRequestHandler method)
log_error()
    (BaseHTTPServer.BaseHTTPRequestHandler method)
log_exception()
    (wsgiref.handlers.BaseHandler method)
log_message()
    (BaseHTTPServer.BaseHTTPRequestHandler method)
log_request()
    (BaseHTTPServer.BaseHTTPRequestHandler method)
log_to_stderr() (in module multiprocessing)
logb() (decimal.Context method)
log10()
    (decimal.Decimal method)
Logger (class in logging)
LoggerAdapter (class in logging)
logging
    Errors
logging (module)
logging.config (module)
logging.handlers (module)
Logical (class in aetypes)
    logical line
    logical_and() (decimal.Decimal method)
    logical_invert() (decimal.Decimal method)
    logical_or() (decimal.Decimal method)
    logical_xor() (decimal.Decimal method)
login() (ftplib.FTP method)
(nntplib.NNTP method)
(poplib.POP3 method)
(tarfile.TarFile method)

LIST_APPEND (opcode)
list_dialects() (in module csv)
list_folders() (mailbox.Maildir method)

(mailbox.MH method)
listallfolders() (mhlib.MH method)
listallsubfolders() (mhlib.MH method)

listdir() (in module dircache)

(in module os)

listen() (asyncore.dispatcher method)

(in module logging.config)

(in module turtle)

(socket.socket method)

Listener (class in multiprocessing.connection)

listfolders() (mhlib.MH method)

listmessages() (mhlib.Folder method)

listMethods() (xmlrpclib.ServerProxy.system method)

ListNoteBook (class in Tix)

lists_folders() (mhlib.MH method)

ListType (in module types)

literal, [1]

literal_eval() (in module ast)

(imaplib.IMAP4 method)

(smtplib.SMTP method)

login_cram_md5() (imaplib.IMAP4 method)

LogRecord (class in logging)

long

built-in function, [1], [2]

integer division

integer literals

long (2to3 fixer)

(built-in class)

long integer

object, [1], [2]

long integer literal

long_info (in module sys)

LONG_MAX, [1]

longMessage (unittest.TestCase attribute)

LongType (in module types)

lookup() (in module codecs)

(in module unicodedat)

(symtable.SymbolTable method)

lookup_error() (in module LookupError)

loop

over mutable sequence statement, [1], [2], [3]

loop control


target

loop() (in module asyncore)

lower() (in module string)

(str method)

lowercase (in module string)
LPAR (in module token)
lseek() (in module os)
Ishift() (in module operator)
LSQB (in module token)
lstat() (in module os)
lstrip() (in module string)
    (str method)
lsub() (imaplib.IMAP4 method)
lr() (in module operator)
    (in module turtle)
Lundh, Fredrik
LWPCookieJar (class in c
M (in module re)
mac_ver() (in module platform)
macerrors (module)
macostools (module)
machine() (in module platform)
MacOS (module)
macpath (module)
macresource (module)
macros (netrc.netrc attribute)
mailbox (module)
Mailbox (class in mailbox)
mailcap (module)
Maildir (class in mailbox)
MaildirMessage (class in mailbox)
MailmanProxy (class in smtpd)
main(), [1]
    (in module py_compile)
    (in module unittest)
mainloop() (FrameWork.Application method)
    (in module turtle)
major() (in module os)
make_archive() (in module distutils.archive_util)
    (in module shutil)
MAKE_CLOSURE (opcode)
make_cookies() (cookielib.CookieJar method)
make_form() (in module fl)
MAKE_FUNCTION (opcode)
make_header() (in module email.header)
make_msgid() (in module email.utils)
methodattrs (2to3 fixer)
methodcaller() (in module xmlrpclib.ServerProxy)
methods (pyclbr.Class attribute)
methodSignature() (xmlrpclib.ServerProxy)
MethodType (in module python)
MH (class in mailbox)
    (class in mhlib)
MHMailbox (class in mailbox)
MHMessage (class in mailbox)
microsecond (datetime.datetime attribute)
    (datetime.time attribute)
MIME base64 encoding
content type
headers, [1]
    quoted-printable encoding
mime_decode_header() (mimify)
mime_encode_header() (mimify)
MIMEApplication (class email.mime.application)
MIMEAudio (class in email.mime.audio)
MIMEBase (class in email.mime.base)
mapfile() (ic.IC method)
    (in module ic)
mapLogRecord()
    (logging.handlers.HTTPHandler method)
**mapping**
    object, [1], [2], [3], [4], [5]
    types, operations on
Mapping (class in collections)
mapping() (msilib.Control method)
MappingView (class in collections)
mapPriority()
    (logging.handlers.SysLogHandler method)
maps() (in module nis)
maptypecreator() (ic.IC method)
    (in module ic)
marshal (module)
**marshalling**
    objects
masking
    operations
match() (in module nis)
    (in module re)
    (re.RegexObject method)
mach_hostname() (in module ssl)
MatchObject (class in re)
**math**
    module, [1]
    math (module)
**max**
    built-in function
max (datetime.date attribute)
    (datetime.datetime attribute)
    (datetime.time attribute)
    (datetime.timedelta attribute)
max() (built-in function)
    (decimal.Context method)
    (decimal.Decimal method)
**MINUS** (in module token)
minus() (decimal.Context method)
    (datetime.datetime attribute)
    (datetime.time attribute)
**MINYEAR** (in module datetime)
mixed() (in module un
misc_header (cmd.Cmd
MissingSectionHeaderError
**MIXERDEV**
mkalias() (in module nis)
mkd() (ftplib.FTP method)
mkdir() (in module os)
mktemp() (in module os)
mknod() (in module os)
mkpath()
    (distutils.ccompiler.CCompiler method)
    (in module distutils.dir_util)
mkstemp() (in module os)
mktemp() (in module os)
mktime() (in module os)
mktime_tz() (in module os)
    (in module rfc822)
mmap (class in mmap)
    (module)
**MMDF** (class in mailbox)
MmdfMailbox (class in mailbox)
**MMDFMessage** (class in mailbox)
mod() (in module operator)
    (io.FileIO attribute)
    (ossaudiodev.oss_audio_device attribute)
    (tarfile.TarInfo attribute)
    (tarfile.TarInfo attribute)
    (turtle.Turtle attribute)
(in module audioop)
MAX_INTERPOLATION_DEPTH (in module ConfigParser)
max_mag() (decimal.Context method)
    (decimal.Decimal method)
maxarray (repr.Repr attribute)
maxdeque (repr.Repr attribute)
maxdict (repr.Repr attribute)
maxDiff (unittest.TestCase attribute)
maxfrozenset (repr.Repr attribute)
maxint (in module sys)
maxlen (collections.deque attribute)
MAXLEN (in module mimify)
maxlevel (repr.Repr attribute)
maxlist (repr.Repr attribute)
maxlong (repr.Repr attribute)
maxother (repr.Repr attribute)
maxpp() (in module audioop)
maxset (repr.Repr attribute)
maxsize (in module sys)
maxstring (repr.Repr attribute)
maxtuple (repr.Repr attribute)
maxunicode (in module sys)
maxval (EasyDialogs.ProgressBar attribute)
MAXYEAR (in module datetime)
MB_ICONASTERISK (in module winsound)
MB_ICONEXCLAMATION (in module winsound)
MB_ICONHAND (in module winsound)
MB_ICONQUESTION (in module winsound)
MB_OK (in module winsound)
mbox (class in mailbox)
mboxMessage (class in mailbox)
md5 (module)
md5() (in module md5)
MemberDescriptorType (in module types)
membership
    test

(robotparser.RobotFileParser)
Modify() (msilib.View method)
modify() (select.epoll method)
(select.poll method)
module
    AL
    CGIHTTPServer
    FrameWork
    SUNAUDIODEV
    SimpleHTTPServer
    __builtin__, [1], [2], [:]
    __main__, [1], [2], [3]
    [7]
    _locale
    array
    base64
    bdb
    binhex
    bsddb, [1], [2], [3]
builtins
cPickle
cmd
compileall
copy
crypt
dbhash
dbm, [1], [2], [3]
dumbdbm
derrno, [1]
exceptions
extension
fcntl
formatter
gdbm, [1], [2]
glob
memmove() (in module ctypes)
MemoryError
MemoryHandler (class in logging.handlers)
memoryview (built-in class)
memset() (in module ctypes)
Menu() (in module FrameWork)
MenuBar() (in module FrameWork)
MenuItem() (in module FrameWork)
merge() (in module heapq)
Message (class in email.message)
    (class in mailbox)
    (class in mhlib)
    (class in mimetools)
    (class in rfc822)
    (in module mimetools)
message digest, MD5, [1]
Message() (in module EasyDialogs)
message_from_file() (in module email)
message_from_string() (in module email)
MessageBeep() (in module winsound)
MessageClass
    (BaseHTTPServer.BaseHTTPRequestHandler attribute)
ErrorMessage
MessageParseError
meta() (in module curses)
meta_path (in module sys)

**metaclass**
    (2to3 fixer)
metavar (optparse.Option attribute)
Meter (class in Tix)
METH_CLASS (built-in variable)
METH_COEXIST (built-in variable)
METH_KEYWORDS (built-in variable)
METH_NOARGS (built-in variable)
METH_O (built-in variable)
METH_OLDARGS (built-in variable)
METH_STATIC (built-in variable)

htmllib
icglue
ihooks
imp
importing
json
knee, [1]
macerrors
mailbox
math, [1]
mimetools
namespace
object, [1], [2]
os, [1]
pickle, [1], [2], [3], [4]
pty
pwd
pyexpat
re, [1], [2]
readline
reexec
rfc822
rlcompleter
search path, [1], [2], [3]
sgmllib
shelve
signal, [1]
sitecustomize
socket, [1]
stat
statvfs
string, [1], [2]
struct
sunaudiodev
METH_VARARGS (built-in variable) method built-in
call object, [1], [2], [3], [4], [5]
user-defined method resolution order

sys, [1], [2], [3], [4], [5]
thread
types
urllib
urlparse
usercustomize
uu
module (pyclbr.Class attribute)
(pyclbr.Function attribute)
module() (in module new)
ModuleFinder (class in modulefinder)
modulefinder (module)
modules (in module sys)
(modulefinder.ModuleFinder attribute)
ModuleType (in module types)
modulo
mono2grey() (in module imageop)
month (datetime.date attribute)
(month.datetime attribute)
month() (in module calendar)
month_abbr (in module calendar)
month_name (in module calendar)
monthcalendar() (in module calendar)
monthdatescalendar() (calendar.Calendar method)
monthdays2calendar() (calendar.Calendar method)
monthdayscalendar() (calendar.Calendar method)
monthrange() (in module calendar)
Morsel (class in Cookie)
mMOST_common() (collections.Counter method)
mouseinterval() (in module curses)
mousemask() (in module curses)
movemove() (curses.panel.Pan
move_file()

distutils.ccompiler.CCompiler

movemessage()

MozillaCookieJar (class MRO

mro() (class method)

msftoframe() (in module msg (httplib.HTTPResponse.

msg) (telnetlib.Telnet.ms

msilib (module)

msvcr (module)

mt_interact() (telnetlib.Telnet mt

time (tarfile.TarInfo attribut

time() (robotparser.Rober

mul() (in module audiooq

(in module operator)

MultiCall (class in xmlrpc)

MultiFile (class in multifil

multifile (module)

MULTILINE (in module re)

MultipartConversionError

multiplication

multiply() (decimal.Context

multiprocessing (module

multiprocessing.connect

multiprocessing.dummy

multiprocessing.Manager

multiprocessing.sharedc
multiprocessing

- multiprocessing.Pool (class in multiprocessing.pool)
- multiprocessing.pool (module in multiprocessing)
- multiprocessing.queues.SimpleQueue (class in multiprocessing)
- multiprocessing.sharedctypes (module)

**mutable**

- object, [1], [2]
- sequence types
- mutable object

**mutable sequence**

- loop over
- object
- MutableMapping (class in collections)
- MutableSequence (class in collections)
- MutableSet (class in collections)
- MutableString (class in UserString)
- mutex (class in mutex)
- mvderwin() (curses.window method)
- mvwin() (curses.window method)
- myrights() (imaplib.IMAP
N

N_TOKENS (in module token)
name, [1], [2]
  binding, [1], [2], [3], [4], [5]
  binding, global
class
function
mangling
rebinding
unbinding
name (cookielib.Cookie attribute)
  (doctest.DocTest attribute)
  (file attribute)
  (in module os)
NAME (in module token)
name (io.FileIO attribute)
  (multiprocessing.Process attribute)
  (ossaudiodev.oss_audio_device attribute)
  (pyclbr.Class attribute)
  (pyclbr.Function attribute)
  (tarfile.TarInfo attribute)
  (threading.Thread attribute)
  (xml.dom.Attr attribute)
  (xml.dom.DocumentType attribute)
name() (in module unicodedata)
namedtuple (in module htmlentitydefs)

named tuple
NamedTemporaryFile() (in module tempfile)
namedtuple() (in module collections)
NameError
  exception

NNTPTemporaryError
  no_proxy, [1]
nocbreak() (in module curses)
  NoDataAllowedErr
Node (class in compiler.ast)
  node() (in module platform)
  nodelay() (curses.window method)
  nodeName (xml.dom.Node attribute)
  NodeTransformer (class in ast)
  noecho() (in module curses)
  NOEXPR (in module locale)
  nofill (htmllib.HTMLParser attribute)
  nok_builtin_names
  (rexec.RExec attribute)
  noautoload() (pickle.Unpickler method)
  NoModificationAllowedErr
  nonblock()
  (ossaudiodev.oss_audio_device method)

None
  object, [1], [2]
None (Built-in object)
  (built-in variable)
NoneType (in module types)
  nonl() (in module curses)
NameError (built-in exception)
namelist() (zipfile.ZipFile method)
nameprep() (in module encodings.idna)

names
  private
namespace, [1]
  global
  module
Namespace (class in argparse)
  (class in multiprocessing.managers)
namespace() (imaplib.IMAP4 method)
Namespace()
  (multiprocessing.managers.SyncManager method)
NAMESPACE_DNS (in module uuid)
NAMESPACE_OID (in module uuid)
NAMESPACE_URL (in module uuid)
NAMESPACE_X500 (in module uuid)
NamespaceErr
namespaceURI (xml.dom.Node attribute)
NaN, [1]
NannyNag
napms() (in module curses)
nargs (optparse.Option attribute)
Nav (module)
Navigation Services
ndiff() (in module difflib)
ndim (memoryview attribute)
ne (2to3 fixer)
ne() (in module operator)
neg() (in module operator)
negation
  nested scope
nested() (in module contextlib)
netrc (class in netrc)
  (module)
NetrcParseError
netscape (cookielib.CookiePolicy
  nonzero (2to3 fixer)
noop() (imaplib.IMAP4 method)
  (poplib.POP3 method)
NoOptionError
NOP (opcode)
norfush() (in module curses)
noraw() (in module curses)
normalize() (decimal.Context method)
  (decimal.Decimal method
   (in module locale)
   (in module unicodedata)
   (xml.dom.Node method)
NORMALIZE_WHITESPACE
  (in module doctest)
normvariate() (in module random)
normcase() (in module os.path)
normpath() (in module os.path)
NoSectionError
NoSuchMailboxError
not
  operator, [1]
not in
  operator, [1], [2]
not_() (in module operator)
NotANumber
notation
notationDecl()
  (xml.sax.handler.DTDHandler method)
NotationDeclHandler()
  (xml.parsers.expat.xmlparser method)
notations
  (xml.dom.DocumentType attribute)
NotConnected
attribute)
Network News Transfer Protocol
new (module)
new() (in module hmac)
    (in module md5)
    (in module sha)
new-style class
new_alignment() (formatter.writer method)
new_compiler() (in module distutils.ccompiler)
new_font() (formatter.writer method)
new_margin() (formatter.writer method)
new_module() (in module imp)
new_panel() (in module curses.panel)
new_spacing() (formatter.writer method)
new_styles() (formatter.writer method)
newconfig() (in module al)
newer() (in module distutils.dep_util)
newer_group() (in module distutils.dep_util)
newer_pairwise() (in module distutils.dep_util)
negroups() (nntplib.NNTP method)
nl
    suppression
NEWLINE (in module token)
NEWLINE token, [1]
newlines (file attribute)
    (io.TextIOBase attribute)
newnews() (nntplib.NNTP method)
nwpad() (in module curses)
nwwin() (in module curses)
next (2to3 fixer)
next() (bsddb.bsddbobject method)
    (built-in function)
    (csv.csvreader method)
    (dbhash.dbhash method)
NoteBook (class in Tix)
Notebook (class in ttk)
NotEmptyError
NOTEQUAL (in module token)
NotFoundError
notify() (threading.Condition method)
notify_all() (threading.Condition method)
notifyAll() (threading.Condition method)
notimeout() (curses.window method)
NotImplemented
    object
NotImplemented (built-in variable)
NotImplementedError
NotImplementedType (in module types)
NotStandaloneHandler()
    (xml.parsers.expat.xmlparser method)
NotSupportedErr
noutrefresh() (curses.window method)
now() (datetime.datetime class method)
NProperty (class in aetypes)
NSIG (in module signal)
nsmallest() (in module heapq)
NT_OFFSET (in module token)
NTEventLogHandler (class in logging.handlers)
ntohl() (in module socket)
ntohs() (in module socket)
ntransfercmd() (ftplib.FTP method)
nnull
(file method)
(generator method)
(iterator method)
(mailbox.oldmailbox method)
(multifile.MultiFile method)
(nntplib.NNTP method)
(tarfile.TarFile method)
(ttk.Treeview method)

next_minus() (decimal.Context method)
   (decimal.Decimal method)
next_plus() (decimal.Context method)
   (decimal.Decimal method)
next_toward() (decimal.Context method)
   (decimal.Decimal method)

nextfile() (in module fileinput)
nextrkey() (in module gdbm)
nexpart() (MimeWriter.MimeWriter method)
nexSibling (xml.dom.Node attribute)
ngettext() (gettext.GNUTranslations method)
   ( gettext.NullTranslations method)
   (in module gettext)
nice() (in module os)
nis (module)
NIST
NL (in module tokenize)
nl() (in module curses)
nl_langinfo() (in module locale)
nlargest() (in module heapq)
nlst() (ftplib.FTP method)

NNTP
   protocol
NNTP (class in nntplib)
NNTPDataExchange
NNTPError
nntplib (module)
NNTPPermanentError
NNTPProtocolError
NNTPReplyError
O_APPEND (in module os)
O_ASYNC (in module os)
O_BINARY (in module os)
O_CREAT (in module os)
O_DIRECT (in module os)
O_DIRECTORY (in module os)
O_DSYNC (in module os)
O_EXCL (in module os)
O_EXLOCK (in module os)
O_NDELAY (in module os)
O_NOATIME (in module os)
O_NOCTTY (in module os)
O_NOFOLLOW (in module os)
O_NOINHERIT (in module os)
O_NONBLOCK (in module os)
O_RANDOM (in module os)
O_RDONLY (in module os)
O_RDWR (in module os)
O_RDONLY (in module os)
O_RDWR (in module os)
O_DSYNC (in module os)
O_SEQUENTIAL (in module os)
O_SHLOCK (in module os)
O_SHORT_LIVED (in module os)
O_SYNC (in module os)
O_TEMPORARY (in module os)
O_TEXT (in module os)
O_TRUNC (in module os)
O_WRONLY (in module os)
object, [1]
    Boolean, [1]
    CObject
    Capsule
    Ellipsis
    None, [1], [2]
    NotImplemented
open_osfhandle() (in module msvcrt)
open_unknown()
(urllib.URLopener method)
OpenDatabase() (in module msilib)
opendir() (in module dircache)
OpenerDirector (class in urllib;
openfolder() (mhlib.MH method)
openfp() (in module sunau)
    (in module wave)
OpenGL
OpenKey() (in module _winreg)
OpenKeyEx() (in module _winreg)
opendir() (in module syslog)
openmessage() (mhlib.Message method)
openmixer() (in module ossaudiodev)
openport() (in module al)
openpty() (in module os)
    (in module pty)
openrf() (in module MacOS)
OpenSSL
    (use in module hashlib)
    (use in module ssl)
OPENSSL_VERSION (in module ssl)
OPENSSL_VERSION_INFO (module ssl)
OPENSSL_VERSION_NUMBER (in module ssl)
OpenView() (msilib.Database method)
Unicode
buffer, [1]
built-in function, [1]
built-in method, [1]
bytearray, [1]
callable, [1]
class, [1], [2], [3], [4]
class instance, [1], [2], [3]
code, [1], [2], [3]
complex
complex number, [1]
deallocation
dictionary, [1], [2], [3], [4], [5], [6], [7]
file, [1], [2], [3], [4]
finalization
floating point, [1], [2]
frame
frozenset, [1]
function, [1], [2], [3], [4], [5]
generator, [1], [2]
immutable, [1], [2]
immutable sequence
instance, [1], [2], [3], [4]
integer, [1], [2]
list, [1], [2], [3], [4], [5], [6], [7], [8]
long integer, [1], [2]
mapping, [1], [2], [3], [4], [5]
method, [1], [2], [3], [4], [5]
module, [1], [2]
mutable, [1], [2]
mutable sequence
numeric, [1], [2], [3], [4]
plain integer
recursive

operation
  Boolean
  binary arithmetic
  binary bitwise
  concatenation
  extended slice
  null
  repetition
  shifting
  slice
  subscript
  unary arithmetic
  unary bitwise

operations
  Boolean, [1]
  bitwise
  masking
  shifting

operations on
  dictionary type
  integer types
  list type
  mapping types
  numeric types
  sequence types, [1]

operator
  !=
  %
  &
  *
  **
  +
  -
  /
  //
sequence, [1], [2], [3], [4], [5], [6], [7], [8]
set, [1], [2], [3]
set type
slice
socket
string, [1], [2], [3], [4]
traceback, [1], [2], [3], [4]
tuple, [1], [2], [3], [4], [5]
type, [1], [2]
unicode
user-defined function, [1], [2]
user-defined method
xrange, [1]
object (built-in class)
    (exceptions.UnicodeError attribute)
object_filenames()
    (distutils.ccompiler.CCompiler method)
objects
    comparing
    flattening
    marshalling
    persistent
    pickling
    serializing
ObjectSpecifier (class in aetypes)
    obufcount()
        (ossaudiodev.oss_audio_device method)
    obuffree()
        (ossaudiodev.oss_audio_device method)
oct
    built-in function
    oct() (built-in function)
        (in module future_builtins)
Open Scripting Architecture
open() (built-in function)
  (FrameWork.DialogWindow method)
  (FrameWork.Window method)
  (distutils.text_file.TextFile method)
  (imaplib.IMAP4 method)
  (in module aifc)
  (in module anydbm)
  (in module cd)
  (in module codecs)
  (in module dbhash)
  (in module dbm)
  (in module dl)
  (in module dumbdbm)
  (in module gdbm)
  (in module gzip)
  (in module io)
  (in module os)
  (in module ossaudiodev)
  (in module posixfile)
  (in module shelf)
  (in module sunau)
  (in module sunaudiodev)
  (in module tarfile)
  (in module wave)
  (in module webbrowser)
  (pipes.Template method)
  (tarfile.TarFile class method)
  (telnetlib.Telnet method)
  (urllib.URLopener method)
  (urllib2.OpenerDirector method)
  (webbrowser.controller method)
  (zipfile.ZipFile method)
open_new() (in module webbrowser)

(output_charset)
  (email.charset.Charset attribute)
output_charset()
  (gettext.NullTranslations method)
output_codec
  (email.charset.Charset attribute)
output_difference()
  (doctest.OutputChecker method)
OutputChecker (class in docte)
OutputString() (Cookie.Morsel method)
OutputType (in module cStringIO)

Overflow (class in decimal)
OverflowError
  (built-in exception), [1],
  [3], [4], [5]
overlay() (curses.window method)

overloading
  operator

Overmars, Mark
overwrite() (curses.window method)
(webbrowser.controller method)
open_new_tab() (in module webbrowser)
  (webbrowser.controller method)
P

P_DETACH (in module os)
P_NOWAIT (in module os)
P_NOWAITO (in module os)
P_OVERLAY (in module os)
P_WAIT (in module os)
pack() (in module aepack)
    (in module struct)
    (mailbox.MH method)
    (struct.Struct method)
pack_array() (xdrlib.Packer method)
pack_bytes() (xdrlib.Packer method)
pack_double() (xdrlib.Packer method)
pack_farray() (xdrlib.Packer method)
pack_float() (xdrlib.Packer method)
pack_fopaque() (xdrlib.Packer method)
pack_fstring() (xdrlib.Packer method)
pack_into() (in module struct)
    (struct.Struct method)
pack_list() (xdrlib.Packer method)
pack_opaque() (xdrlib.Packer method)
pack_string() (xdrlib.Packer method)
package, [1], [2]
package variable
    __all__
    Packer (class in xdrlib)
packevent() (in module aetools)
packing
    binary data
packing (widgets)
PAGER, [1]
pair_content() (in module curses)
pair_number() (in module curses)
PanedWindow (class in Tix)
parameter

PyExc_Exception
PyExc_FloatingPointError
PyExc_ImportError
PyExc_IndexError
PyExc_IOError
PyExc_KeyboardInterrupt
PyExc_KeyError
PyExc_LookupError
PyExc_MemoryError
PyExc_NameError
PyExc_NotImplemented
PyExc_OverflowError
PyExc_RuntimeError
PyExc_SyntaxError
PyExc_SystemError
PyExc_SystemExit
PyExc_TypeError
PyExc_ValueError
PyExc_ZeroDivisionError
pyexpat

PyFile_AsFile (C function)
PyFile_Check (C function)
PyFile_DecUseCount (C function)
PyFile_FromFile (C function)
PyFile_GetLine (C function)
PyFile_IncUseCount (C function)
PyFile_Name (C function)
PyFile_SetBufSize (C function)
call semantics
difference from argument
function definition
value, default
pardir (in module os)
paren (2to3 fixer)
parent (urllib2.BaseHandler attribute)
parent() (ttk.Treeview method)
parenthesized form
parentNode (xml.dom.Node attribute)
paretovariate() (in module random)
parse() (doctest.DocTestParser method)
   (email.parser.Parser method)
   (in module ast)
   (in module cgi)
   (in module compiler)
   (in module xml.dom.minidom)
   (in module xml.dom.pulldom)
   (in module xml.etree.ElementTree)
   (in module xml.sax)
   (robotparser.RobotFileParser method)
   (string.Formatter method)
   (xml.etree.ElementTree.ElementTree method)
Parse() (xml.parsers.expat.xmlparser method)
parse() (xml.sax.xmlreader.XMLReader method)
parse_and_bind() (in module readline)
parse_args() (argparse.ArgumentParser method)
PARSE_COLNAMES (in module sqlite3)
parse_config_h() (in module sysconfig)
PARSE_DECLTYPES (in module sqlite3)
parse_header() (in module cgi)
parse_known_args() (argparse.ArgumentParser method)
parse_multipart() (in module cgi)
parse_qs() (in module cgi)
   (in module urlparse)
parse_qsl() (in module cgi)
PyFile_SetEncoding
PyFile_SetEncodingAndErrors
PyFile_SoftSpace
PyFile_Type (C variable)
PyFile_WriteObject
PyFile_WriteString
PyFloat_AsDouble
PyFloat_AsString
PyFloat_Check
PyFloat_CheckExact
PyFloat_ClearFreeList
PyFloat_FromDouble
PyFloat_FromString
PyFloat_GetInfo
PyFloat_GetMax
PyFloat_GetMin
PyFloat_Type (C variable)
PyFloatObject (C type)
PyFrame_GetLineNumber
PyFrozenSet_Check
PyFrozenSet_CheckExact
PyFrozenSet_Type
PyFunction_Check
PyFunction_GetClosure
PyFunction_GetCode
PyFunction_GetDefaults
PyFunction_GetGlobals
PyFunction_GetModule
PyFunction_New
PyFunction_SetClosure
PyFunction_SetDefaults
PyFunction_Type
PyFunctionObject (C type)
PYFUNCTYPE (in module ctypes)
PyGen_Check
(in module urlparse)
parseaddr() (in module email.utils)
    (in module rfc822)
parsedate() (in module email.utils)
    (in module rfc822)
parsedate_tz() (in module email.utils)
    (in module rfc822)
parseFile() (in module compiler)
ParseFile() (xml.parsers.expat.xmlparser method)
ParseFlags() (in module imaplib)
parser
Parser (class in email.parser)
parser (module)
ParserCreate() (in module xml.parsers.expat)
ParserError
ParseResult (class in urlparse)
parsesequence() (mhlib.Folder method)
parsestr() (email.parser.Parser method)
parseString() (in module xml.dom.minidom)
    (in module xml.dom.pulldom)
    (in module xml.sax)
parseurl() (ic.IC method)
    (in module ic)
parsing
   Python source code
   URL
ParsingError
partial() (imaplib.IMAP4 method)
    (in module functools)
partition() (str method)
Pascal
   language
pass
   statement
pass_() (poplib.POP3 method)
Paste
PATH, [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12],
path

configuration file
module search, [1], [2], [3], [4], [5], [6]
operations

path (BaseHTTPServer.BaseHTTPRequestHandler
attribute)
    (cookielib.Cookie attribute)
    (in module sys), [1], [2], [3]
Path browser
path_hooks (in module sys)
path_importer_cache (in module sys)
path_return_ok() (cookielib.CookiePolicy method)
pathconf() (in module os)
pathconf_names (in module os)
pathname2url() (in module urllib)
pathsep (in module os)
pattern (re.RegexObject attribute)
pause() (in module signal)
PAUSED (in module cd)
PAX_FORMAT (in module tarfile)
pax_headers (tarfile.TarFile attribute)
    (tarfile.TarInfo attribute)
pbkdf2_hmac() (in module hashlib)
pd() (in module turtle)
Pdb (class in pdb), [1]
pdb (module)
peek() (io.BufferedReader method)
PEM_cert_to_DER_cert() (in module ssl)
pen() (in module turtle)
pencolor() (in module turtle)
PendingDeprecationWarning
pendown() (in module turtle)
pensize() (in module turtle)
penup() (in module turtle)
PERCENT (in module token)
PERCENTEQUAL (in module token)
Performance
permutations() (in module itertools)
Persist() (msilib.SummaryInformation method)
persistence
persistent
  objects
persistent_id (pickle protocol)
persistent_load (pickle protocol)
pformat() (in module pprint)
    (pprint.PrettyPrinter method)
phase() (in module cmath)
Philbrick, Geoff
physical line, [1], [2]
pi (in module cmath)
    (in module math)
pick() (in module gl)
pickle
  module, [1], [2], [3], [4]
pickle (module)
pickle() (in module copy_reg)
PickleError
Pickler (class in pickle)
pickletools (module)
pickling
  objects
PicklingError
pid (multiprocessing.Process attribute)
  (popen2.Popen3 attribute)
  (subprocess.Popen attribute)
PIL (the Python Imaging Library)
PIPE (in module subprocess)
Pipe() (in module multiprocessing)
pipe() (in module os)
PIPE_BUF (select.select attribute)
pipes (module)
PixMapWrapper (module)
PKG_DIRECTORY (in module imp)
pkgutil (module)
plain integer
object
plain integer literal
PLAT
platform (in module sys), [1]
  (module)
platform() (in module platform)
PLAYING (in module cd)
PlaySound() (in module winsound)
plist
  file
plistlib (module)
plock() (in module os)
plus
PLUS (in module token)
plus() (decimal.Context method)
PLUSEQUAL (in module token)
pm() (in module pdb)
pnum (in module cd)
POINTER() (in module ctypes)
pointer() (in module ctypes)
polar() (in module cmath)
poll() (in module select)
  (multiprocessing.Connection method)
  (popen2.Popen3 method)
  (select.epoll method)
  (select.poll method)
  (subprocess.Popen method)
pop() (array.array method)
  (asynchat.fifo method)
  (collections.deque method)
  (dict method)
  (list method)
  (mailbox.Mailbox method)
  (multifile.MultiFile method)
  (set method)
POP3
  protocol
POP3 (class in poplib)
POP3_SSL (class in poplib)
pop_alignment() (formatter.formatter method)
POP_BLOCK (opcode)
pop_font() (formatter.formatter method)
POP_JUMP_IF_FALSE (opcode)
POP_JUMP_IF_TRUE (opcode)
pop_margin() (formatter.formatter method)
pop_source() (shlex.shlex method)
pop_style() (formatter.formatter method)
POP_TOP (opcode)
Popen (class in subprocess)
popen() (in module os), [1], [2]
    (in module platform)
popen2 (module)
popen2() (in module os)
    (in module popen2)
Popen3 (class in popen2)
popen3() (in module os)
    (in module popen2)
Popen4 (class in popen2)
popen4() (in module os)
    (in module popen2)
popitem() (collections.OrderedDict method)
    (dict method)
        (mailbox.Mailbox method)
popleft() (collections.deque method)
poplib (module)
PopupMenu (class in Tix)
port (cookielib.Cookie attribute)
port_specified (cookielib.Cookie attribute)
PortableUnixMailbox (class in mailbox)
pos (re.MatchObject attribute)
pos() (in module operator)
    (in module turtle)
position() (in module turtle)
**positional argument**
POSIX
I/O control

file object

threads

posix (module)

(tarfile.TarFile attribute)

posixfile (module)

POSIXLY_CORRECT

post() (ntplib.NNTP method)

(ossaudiodev.oss_audio_device method)

post_mortem() (in module pdb)

postcmd() (cmd.Cmd method)

postloop() (cmd.Cmd method)

pow

    built-in function, [1], [2], [3], [4], [5]

pow() (built-in function)

    (in module math)

    (in module operator)

power() (decimal.Context method)

pprint (module)

pprint() (bdb.Breakpoint method)

    (in module pprint)

    (pprint.PrettyPrinter method)

prcal() (in module calendar)

preamble (email.message.Message attribute)

precedence

    operator

precmd() (cmd.Cmd method)

prefix, [1], [2], [3]

PREFIX (in module distutils.sysconfig)

prefix (in module sys)

    (xml.dom.Attr attribute)

    (xml.dom.Node attribute)

    (zipimport.zipimporter attribute)

PREFIXES (in module site)

preloop() (cmd.Cmd method)

preorder() (compiler.visitor.ASTVisitor method)

prepare_input_source() (in module xml.sax.saxutils)
prepend() (pipes.Template method)
preprocess() (distutils.ccompiler.CCompiler method)
PrettyPrinter (class in pprint)
prev() (ttk.Treeview method)
previous() (bsddb.bsddbobject method)
  (dbhash.dbhash method)
previousSibling (xml.dom.Node attribute)
primary
print
  statement, [1], [2]
print (2to3 fixer)
print() (built-in function)
Print() (in module findertools)
print_callees() (pstats.Stats method)
print_callers() (pstats.Stats method)
print_directory() (in module cgi)
print_environ() (in module cgi)
print_environ_usage() (in module cgi)
print_exc() (in module traceback)
  (timeit.Timer method)
print_exception() (in module traceback)
PRINT_EXPR (opcode)
print_form() (in module cgi)
print_help() (argparse.ArgumentParser method)
PRINT_ITEM (opcode)
PRINT_ITEM_TO (opcode)
print_last() (in module traceback)
PRINT_NEWLINE (opcode)
PRINT_NEWLINE_TO (opcode)
print_stack() (in module traceback)
print_stats() (profile.Profile method)
  (pstats.Stats method)
print_tb() (in module traceback)
print_usage() (argparse.ArgumentParser method)
  (optparse.OptionParser method)
print_version() (optparse.OptionParser method)
printable (in module string)
printdir() (zipfile.ZipFile method)
printf-style formatting
PriorityQueue (class in Queue)
private
  names
prmonth() (calendar.TextCalendar method)
  (in module calendar)
procedure
call
process
  group, [1]
  id
  id of parent
  killing, [1]
  signalling, [1]
Process (class in multiprocessing)
process() (logging.LoggerAdapter method)
process_message() (smtpd.SMTPServer method)
process_request() (SocketServer.BaseServer method)
processes, light-weight
processfile() (in module gensuitemodule)
processfile_fromresource() (in module gensuitemodule)
ProcessingInstruction() (in module xml.etree.ElementTree)
  (xml.sax.handler.ContentHandler method)
  (xml.parsers.expat.xmlparser method)
  (xml.sax.handler.ContentHandler method)
  (xml.parsers.expat.xmlparser method)
  (xml.sax.handler.ContentHandler method)
  (xml.parsers.expat.xmlparser method)
  (xml.parsers.expat.xmlparser method)
  (xml.parsers.expat.xmlparser method)
  (xml.parsers.expat.xmlparser method)
  (xml.parsers.expat.xmlparser method)
  processor time
  processor() (in module platform)
  product() (in module itertools)
Profile (class in hotshot)
  (class in profile)
profile (module)
profile function, [1], [2]
profiler, [1]
profiling, deterministic
Process (class in multiprocessing)
process() (logging.LoggerAdapter method)
program
Progressbar (class in ttk)
ProgressBar() (in module EasyDialogs)
prompt (cmd.Cmd attribute)
prompt_user_passwd() (urllib.FancyURLopener method)
prompts, interpreter
propagate (logging.Logger attribute)
property (built-in class)
property list
property_declaration_handler (in module xml.sax.handler)
property_dom_node (in module xml.sax.handler)
property_lexical_handler (in module xml.sax.handler)
property_xml_string (in module xml.sax.handler)
prot_c() (ftplib.FTP_TLS method)
prot_p() (ftplib.FTP_TLS method)
proto (socket.socket attribute)
protocol
  CGI
  FTP, [1]
  HTTP, [1], [2], [3], [4]
  IMAP4
  IMAP4_SSL
  IMAP4_stream
  NNTP
  POP3
  SMTP
  Telnet
  context management
  iterator
  protocol (ssl.SSLContext attribute)
  PROTOCOL_SSLv2 (in module ssl)
  PROTOCOL_SSLv23 (in module ssl)
  PROTOCOL_SSLv3 (in module ssl)
  PROTOCOL_TLS (in module ssl)
  PROTOCOL_TLSv1 (in module ssl)
PROTOCOL_TLSv1_1 (in module ssl)
PROTOCOL_TLSv1_2 (in module ssl)
protocol_version
(BaseHTTPServer.BaseHTTPRequestHandler attribute)
PROTOCOL_VERSION (imaplib.IMAP4 attribute)
ProtocolError (class in xmlrpclib)
proxy() (in module weakref)
proxyauth() (imaplib.IMAP4 method)
ProxyBasicAuthHandler (class in urllib2)
ProxyDigestAuthHandler (class in urllib2)
ProxyHandler (class in urllib2)
ProxyType (in module weakref)
ProxyTypes (in module weakref)
prstr() (in module fm)
pryear() (calendar.TextCalendar method)
ps1 (in module sys)
ps2 (in module sys)
pstats (module)
ptime (in module cd)
pty
  module
pty (module)
pu() (in module turtle)
publicId (xml.dom.DocumentType attribute)
PullDOM (class in xml.dom.pulldom)
punctuation (in module string)
PureProxy (class in smtpd)
purge() (in module re)
Purpose.CLIENT_AUTH (in module ssl)
Purpose.SERVER_AUTH (in module ssl)
push() (asyncchat.async_chat method)
  (asyncchat.fifo method)
  (code.InteractiveConsole method)
  (multifile.MultiFile method)
push_alignment() (formatter.formatter method)
push_font() (formatter.formatter method)
push_margin() (formatter.formatter method)
push_source() (shlex.shlex method)
push_style() (formatter.formatter method)
push_token() (shlex.shlex method)
push_with_producer() (asyncio.async_producer method)
pushbutton() (msilib.Dialog method)
put()
(multiprocessing.multiprocessing.queues.SimpleQueue method)
(Queue.Queue method)
(multiprocessing.Queue method)
put_nowait() (multiprocessing.Queue method)
(Queue.Queue method)
putch() (in module msvcrt)
putenv() (in module os)
putheader() (http.client.HTTPConnection method)
putp() (in module curses)
putrequest() (http.client.HTTPConnection method)
putsequences() (mhlib.Folder method)
putwch() (in module msvcrt)
putwin() (curses.window method)
pwd
   module
pwd (module)
pwd() (ftp.client.FTP method)
pwlcurve() (in module gl)
py3kwarning (in module sys)
Py_AddPendingCall (C function)
Py_AddPendingCall()
Py_AtExit (C function)
Py_BEGIN_ALLOW_THREADS
   (C macro)
Py_BLOCK_THREADS (C macro)
Py_buffer (C type)
Py_buffer.buf (C member)
Py_buffer.internal (C member)
Py_buffer.itemsize (C member)
Py_buffer.ndim (C member)

Python 3000
Python Editor
Python Enhancement
PEP 100
PEP 11
PEP 205, [1
Py_buffer.readonly (C member) (PEP 207)
Py_buffer.shape (C member) (PEP 208)
Py_buffer.strides (C member) (PEP 217)
Py_buffer.suboffsets (C member) (PEP 218, [1]
Py_BuildValue (C function) (PEP 218, [1]
Py_CLEAR (C function) (PEP 227, [1]
py_compile (module) (PEP 229)
PY_COMPILED (in module imp) (PEP 230, [1]
Py_CompileString (C function) (PEP 232)
Py_CompileString(), [1], [2] (PEP 234)
Py_CompileStringFlags (C function) (PEP 236, [1]
Py_complex (C type) (PEP 237, [1]
Py_DECREF (C function) (PEP 238, [1]
Py_DECREF() (PEP 241)
Py_END_ALLOW_THREADS (C macro) (PEP 243)
Py_END_OF_BUFFER (C variable) (PEP 249, [1]
Py_EndInterpreter (C function) (PEP 252, [1]
Py_EnterRecursiveCall (C function) (PEP 253, [1]
Py_eval_input (C variable) (PEP 255, [1]
Py_Exit (C function) (PEP 261, [1]
Py_False (C variable) (PEP 263, [1]
Py_FatalError (C function) (PEP 264)
Py_FatalError() (PEP 273, [1]
Py_FdIsInteractive (C function) (PEP 275)
Py_file_input (C variable) (PEP 277)
Py_Finalize (C function) (PEP 278, [1]
Py_Finalize(), [1], [2], [3], [4] (PEP 279)
Py_FindMethod (C function) (PEP 282, [1]
PY_FROZEN (in module imp) (PEP 283)
Py_GetBuildInfo (C function) (PEP 285, [1]
Py_GetCompiler (C function) (PEP 288)
Py_GetCopyright (C function) (PEP 289, [1]
Py_GetExecPrefix (C function) (PEP 292, [1]
Py_GetExecPrefix() (PEP 293)
Py_GetPath (C function) (PEP 3000)
Py_GetPath(), [1] (PEP 3000)
Py_GetPlatform (C function) (PEP 301, [1]
Py_GetPrefix (C function)
Py_GetPrefix()
Py_GetProgramFullPath (C function)
Py_GetProgramFullPath()
Py_GetProgramName (C function)
Py_GetPythonHome (C function)
Py_GetVersion (C function)
Py_INCREF (C function)
Py_INCREF()
Py_Initialize (C function)
Py_Initialize(), [1], [2], [3]
Py_InitializeEx (C function)
Py_InitModule (C function)
Py_InitModule3 (C function)
Py_InitModule4 (C function)
Py_IsInitialized (C function)
Py_IsInitialized()
Py_LeaveRecursiveCall (C function)
Py_Main (C function)
Py_NewInterpreter (C function)
Py_None (C variable)
py_object (class in ctypes)
PY_PRINT_RAW
Py_REFCNT (C macro)
Py_RETURN_FALSE (C macro)
Py_RETURN_NONE (C macro)
Py_RETURN_TRUE (C macro)
Py_SetProgramName (C function)
Py_SetProgramName(), [1], [2], [3]
Py_SetPythonHome (C function)
Py_single_input (C variable)
Py_SIZE (C macro)
PY_SOURCE (in module imp)
PY_SSIZE_T_MAX
py_suffix_importer() (in module imputil)
Py_TPFLAGS_BASETYPE (built-in variable)
Py_TPFLAGS_CHECKTYPES (built-in variable)
Py_TPFLAGS_DEFAULT (built-in variable)
Py_TPFLAGS_GC (built-in variable)
Py_TPFLAGS_HAVE_CLASS (built-in variable)
Py_TPFLAGS_HAVE_GC (built-in variable)  PEP 341
Py_TPFLAGS_HAVE_GETCHARBUFFER (built-in variable), [1]  PEP 342, [1
Py_TPFLAGS_HAVE_INPLACEOPS (built-in variable)  PEP 343, [1
Py_TPFLAGS_HAVE_ITER (built-in variable)  PEP 347
Py_TPFLAGS_HAVE_RICHCOMPARE (built-in variable)  PEP 352, [1
Py_TPFLAGS_HAVE_SEQUENCE_IN (built-in variable)  PEP 353, [1
Py_TPFLAGS_HAVE_WEAKREFS (built-in variable)  PEP 356
Py_TPFLAGS_HEAPTYPE (built-in variable)  PEP 357
Py_TPFLAGS_READY (built-in variable)  PEP 361
Py_TPFLAGS_READYING (built-in variable)  PEP 366
Py_tracefunc (C type)  PEP 370, [1
Py_True (C variable)  PEP 371
Py_TYPE (C macro)  PEP 372
Py_UNBLOCK_THREADS (C macro)  PEP 373
Py_UNICODE (C type)  PEP 378, [1
Py_UNICODE_ISALNUM (C function)  PEP 389
Py_UNICODE_ISALPHA (C function)  PEP 391
Py_UNICODE_ISDECIMAL (C function)  PEP 427
Py_UNICODE_ISDIGIT (C function)  PEP 434
Py_UNICODE_ISLINEBREAK (C function)  PEP 453, [1
Py_UNICODE_ISLOWER (C function)  PEP 466, [1
Py_UNICODE_ISNUMERIC (C function)  PEP 476, [1
Py_UNICODE_ISSPACE (C function)  PEP 477, [1
Py_UNICODE_ISTITLE (C function)  PEP 493, [1
Py_UNICODE_ISSUPPER (C function)  PEP 5, [1
Py_UNICODE_TODECIMAL (C function) PEP 6
Py_UNICODE_TODIGIT (C function) PEP 8, [1, [1
Py_UNICODE_TOLOWER (C function) Python Imaging Library
Py_UNICODE_TONUMERIC (C function) Python Package Index (PyPI)
Py_UNICODE_TOTITLE (C function) .pypirc file
Py_UNICODE_TOUPPER (C function) PYTHON*
Py_VaBuildValue (C function) python_branch()
Py_VISIT (C function) python_build()
Py_XDECREF (C function) python_compile
Py_XDECREF() PYTHON_DOM
PyCode_Check (C function)          PyTypeObject.t
PyCode_GetNumFree (C function)     PyTypeObject.t
PyCode_New (C function)            PyTypeObject.t
PyCode_NewEmpty (C function)       PyTypeObject.t
PyCode_Type (C variable)           PyTypeObject.t
PyCodec_BackslashReplaceErrors (C function) PyTypeObject.t
PyCodec_Decode (C function)         PyTypeObject.t
PyCodec_Decoder (C function)        PyTypeObject.t
PyCodec_Encode (C function)         PyTypeObject.t
PyCodec_IgnoreErrors (C function)   PyTypeObject.t
PyCodec_IncrementalDecoder (C function) PyTypeObject.t
PyCodec_IncrementalEncoder (C function) PyTypeObject.t
PyCodecKNOWNEncoding (C function)   PyTypeObject.t
PyCodec.LookupError (C function)    PyTypeObject.t
PyCodec_Register (C function)       PyTypeObject.t
PyCodec_RegisterError (C function)  PyTypeObject.t
PyCodec_ReplaceErrors (C function)  PyTypeObject.t
PyCodec_StreamReader (C function)   PyTypeObject.t
PyCodec_StreamWriter (C function)   PyTypeObject.t
PyCodec_StrictErrors (C function)   PyTypeObject.t
PyCodec_XMLCharRefReplaceErrors (C function) PyTypeObject.t
PyCodeObject (C type)              PyTypeObject.t
PyCompileError                     PyTypeObject.t
PyCompilerFlags (C type)            PyTypeObject.t
PyComplex_AsCComplex (C function)   PyTypeObject.t
PyComplex_Check (C function)        PyTypeObject.t
PyComplex_CheckExact (C function)   PyTypeObject.t
PyComplex_FromCComplex (C function) PyTypeObject.t
PyComplex_FromDoubles (C function)  PyTypeObject.t
PyComplex_ImagAsDouble (C function) PyTypeObject.t
PyComplex_RealAsDouble (C function) PyTypeObject.t
PyComplex_Type (C variable)         PyTypeObject.t
PyComplexObject (C type)            PyTypeObject.t
PyDate_Check (C function)           PyTypeObject.t
PyDate_CheckExact (C function)      PyTypeObject.t
PyDate_FromDate (C function)        PyTypeObject.t
PyDate_FromTimestamp (C function)   PyTypeObject.t
PyDateTime_Check (C function)       PyTypeObject.t
PyDict_Next (C function)
PyDict_SetItem (C function)
PyDict_SetItemString (C function)
PyDict_Size (C function)
PyDict_Type (C variable)
PyDict_Update (C function)
PyDict.Values (C function)
PyDictObject (C type)
PyDictProxy_New (C function)
PyDLL (class in ctypes)
pydoc (module)
PyErr_BadArgument (C function)
PyErr_BadInternalCall (C function)
PyErr_CheckSignals (C function)
PyErr_Clear (C function)
PyErr_Clear(), [1]
PyErr_ExceptionMatches (C function)
PyErr_ExceptionMatches()  
PyErr_Fetch (C function)
PyErr_Fetch()  
PyErr_Format (C function)
PyErr_GivenExceptionMatches (C function)
PyErr_NewException (C function)
PyErr_NewExceptionWithDoc (C function)
PyErr_NoMemory (C function)
PyErr_NormalizeException (C function)
PyErr_Occurred (C function)
PyErr_Occurred()  
PyErr_Print (C function)
PyErr_PrintEx (C function)
PyErr_Restore (C function)
PyErr_Restore()  
PyErr_SetExcFromWindowsErr (C function)
PyErr_SetExcFromWindowsErrWithFilename (C function)
PyErr_SetExcFromWindowsErrWithFilenameObject (C function)
PyErr_SetFromErrno (C function)
PyErr_SetFromErrnoWithFilename (C function)
PyErr_SetFromErrnoWithFilenameObject (C function)
PyErr_SetFromWindowsErr (C function)
PyErr_SetFromWindowsErrWithFilename (C function)
PyErr_SetFromWindowsErrWithFilenameObject (C function)
PyErr_SetInterrupt (C function)
PyErr_SetNone (C function)
PyErr_SetObject (C function)
PyErr_SetString (C function)
PyErr_Warn (C function)
PyErr_WarnEx (C function)
PyErr_WarnExplicit (C function)
PyErr_WarnPy3k (C function)
PyErr_WriteUnraisable (C function)
PyEval_AcquireLock (C function)
PyEval_AcquireLock()
PyEval_AcquireThread (C function)
PyEval_EvalCode (C function)
PyEval_EvalCodeEx (C function)
PyEval_EvalFrame (C function)
PyEval_EvalFrameEx (C function)
PyEval_GetBuiltins (C function)
PyEval_GetCallStats (C function)
PyEval_GetFrame (C function)
PyEval_GetFuncDesc (C function)
PyEval_GetFuncName (C function)
PyEval_GetGlobals (C function)
PyEval_GetLocals (C function)
PyEval_GetRestricted (C function)
PyEval_InitThreads (C function)
PyEval_InitThreads()
PyEval_MergeCompilerFlags (C function)
PyEval_ReInitThreads (C function)
PyEval_ReleaseLock (C function)
PyEval_ReleaseLock(), [1]
PyEval_ReleaseThread (C function)
PyEval_ReleaseThread()
PyEval_RestoreThread (C function)
PyUnicode_Join (C function)
PyUnicode_Replace (C function)
PyUnicode_RichCompare (C function)
PyUnicode_Split (C function)
PyUnicode_Splitlines (C function)
PyUnicode_Tailmatch (C function)
PyUnicode_Translate (C function)
PyUnicode_TranslateCharmap (C function)
PyUnicode_Type (C variable)
PyUnicodeDecodeError_Create (C function)
PyUnicodeDecodeError_GetEncoding (C function)
PyUnicodeDecodeError_GetEnd (C function)
PyUnicodeDecodeError_GetObject (C function)
PyUnicodeDecodeError_GetReason (C function)
PyUnicodeDecodeError_GetStart (C function)
PyUnicodeDecodeError_SetEnd (C function)
PyUnicodeDecodeError_SetReason (C function)
PyUnicodeDecodeError_SetStart (C function)
PyUnicodeEncodeError_Create (C function)
PyUnicodeEncodeError_GetEncoding (C function)
PyUnicodeEncodeError_GetEnd (C function)
PyUnicodeEncodeError_GetObject (C function)
PyUnicodeEncodeError_GetReason (C function)
PyUnicodeEncodeError_GetStart (C function)
PyUnicodeEncodeError_SetEnd (C function)
PyEval_RestoreThread(), [1]  PyUnicodeEncodeError_SetReason (C function)
PyEval_SaveThread (C function)  PyUnicodeEncodeError_SetStart (C function)
PyEval_SaveThread(), [1]  PyUnicodeEncodeError (C function)
PyEval_SetProfile (C function)  PyUnicodeObject (C type)
PyEval_SetTrace (C function)  PyUnicodeTranslateError_Create (C function)
PyEval_ThreadsInitialized (C function)  PyUnicodeTranslateError_GetEnd (C function)
PyExc_ArithmeticError  PyUnicodeTranslateError_GetObject (C function)
PyExc_AssertionError  PyUnicodeTranslateError_GetReason (C function)
PyExc_AttributeError  PyUnicodeTranslateError_GetStart (C function)
PyExc_BaseException  PyUnicodeTranslateError_SetEnd (C function)
PyExc_environmentError  PyUnicodeTranslateError_SetReason (C function)
PyExc_EOFError  PyUnicodeTranslateError_SetStart (C function)
PyVarObject (C type)  PyVarObject.ob_size (C member)
PyVarObject_HEAD_INIT (C macro)
PyWeakref_Check (C function)  PyWeakref_CheckProxy (C function)
PyWeakref_CheckRef (C function)  PyWeakref_GET_OBJECT (C function)
PyWeakref_GetObject (C function)  PyWeakref_NewProxy (C function)
PyWeakref_NewRef (C function)  PyWrapper_New (C function)
PyZipFile (class in zipfile)
qdevice() (in module fl)
QDPoint (class in aetypes)
QDRectangle (class in aetypes)
qflush() (in module curses)
QName (class in xml.etree.ElementTree)
qread() (in module fl)
qreset() (in module fl)
qsize() (multiprocessing.Queue method)
  (Queue.Queue method)
qtest() (in module fl)
quantize() (decimal.Context method)
  (decimal.Decimal method)
QueryInfoKey() (in module _winreg)
queryparams() (in module al)
QueryReflectionKey() (in module _winreg)
QueryValue() (in module _winreg)
QueryValueEx() (in module _winreg)
Queue (class in multiprocessing)
Queue() (multiprocessing.managers.SyncManager method)
quick_ratio() (difflib.SequenceMatcher method)
quit (built-in variable)
  (ftplib.FTP method)
  (ntplib.NNTP method)
  (poplib.POP3 method)
  (smtplib.SMTP method)
quopri (module)
quote() (in module email.utils)
  (in module pipes)
  (in module rfc822)
  (in module urllib)
QUOTE_ALL (in module csv)
QUOTE_MINIMAL (in module csv)
QUOTE_NONE (in module csv)
QUOTE_NONNUMERIC (in module csv)
quote_plus() (in module urllib)
quoteattr() (in module xml.sax.saxutils)
quotechar (csv.Dialect attribute)
quoted-printable encoding
quotes
  backward, [1]
  reverse, [1]
quotes (shlex.shlex attribute)
quoting (csv.Dialect attribute)
(class in Queue)
(module)
queue (sched.scheduler attribute)
r_eval() (rexec.RExec method)
r_exec() (rexec.RExec method)
r_execfile() (rexec.RExec method)
r_import() (rexec.RExec method)
R_OK (in module os)
r_open() (rexec.RExec method)
r_reload() (rexec.RExec method)
r_unload() (rexec.RExec method)
radians() (in module math)
   (in module turtle)
RadioButtonGroup (class in msilib)
radiogroup() (msilib.Dialog method)
radix() (decimal.Context method)
   (decimal.Decimal method)
RADIXCHAR (in module locale)
raise
   statement, [1]
raise (2to3 fixer)
raise an exception
RAISE_VARARGS (opcode)
raising
   exception
RAND_add() (in module ssl)
RAND_eggd() (in module ssl)
RAND_status() (in module ssl)
randint() (in module random)
random (module)
random() (in module random)
randrange() (in module random)
r
   built-in function
Range (class in aetypes)
range() (built-in function)
ratecv() (in module audioop)
ratio() (difflib.SequenceMatcher method)
renames() (in module os)
reorganize() (in module gdbm)
repeat() (in module itertools)
   (in module timeit)
   (timeit.Timer method)
repetition
   operation
replace() (curses)
   (datetime.datetime method)
   (datetime.datetime method)
   (in module timeit)
replace(2to3 fixer)
replace an exception
replace Child() (modulefinder).
report() (filecmp.dircmp)
   modulefinder
REPORT_CDIFF (in module doctest)
report_failure() (doctest.DocTestRunner)
report_history() (doctest.DocTestRunner)
report_partial_closure() (filecmp.dircmp)
REPORT_NDIFF (in module doctest)
report_partial_closure() (filecmp.dircmp)
REPORT_ONLY (in module doctest)
report_partial_closure() (filecmp.dircmp)
REPORT_UDIFF (in module doctest)
report_unbalanced() (sgmllib.SGMLParser)
report_unexpected_exception() (doctest.DocTestRunner)
Rational (class in numbers)
raw (io.BufferedIOBase attribute)
raw input
raw string
raw() (in module curses)
raw_decode() (json.JSONDecoder method)
raw_input
  built-in function, [1]
raw_input (2to3 fixer)
raw_input() (built-in function)
  (code.InteractiveConsole method)
RawArray() (in module multiprocessing.sharedctypes)
RawConfigParser (class in ConfigParser)
RawDescriptionHelpFormatter (class in argparse)
RawIOBase (class in io)
RawPen (class in turtle)
RawTextHelpFormatter (class in argparse)
RawTurtle (class in turtle)
RawValue() (in module multiprocessing.sharedctypes)
RBRACE (in module token)
re
  module, [1], [2]
re (module)
  (re.MatchObject attribute)
read() (array.array method)
  (ConfigParser.RawConfigParser method)
  (bz2.BZ2File method)
  (chunk.Chunk method)
  (codecs.StreamReader method)
  (file method)
  (httplib.HTTPResponse method)
  (imaplib.IMAP4 method)
  (in module imgfile)
  (in module os)
  (io.BufferedIOBase method)
  (io.BufferedReader method)
  (io.RawIOBase method)
REPORTING_FLAGS (in module doctest)
repr
  built-in function
  repr (2to3 fixer)
  Repr (class in repr)
  repr() (built-in function)
  (in module repr)
  (repr.Repr method)
  repr1() (repr.Repr method)
representation
  integer
Request (class in urllib2)
request()
  (httplib.HTTPConnection method)
request_queue_size
  (SocketServer.BaseServer attribute)
request_uri()
  (in module wsgiref.util)
request_version
  (BaseHTTPServer.BaseHTTPRequestHandler attribute)
RequestHandlerClass
  (SocketServer.BaseServer attribute)
requires()
  (in module test.test_support)
reserved
  (zipfile.ZipInfo attribute)
  reserved word
RESERVED_FUTURE (in module uuid)
RESERVED_MICROSOFT (in module uuid)
RESERVED_NCS (in module uuid)
reset()
  (bdb.Bdb method)
  (HTMLParser.HTMLParser method)
  (codecs.IncrementalDecoder method)
  (codecs.IncrementalEncoder method)
  (codecs.StreamReader method)
  (codecs.StreamWriter method)
  (osaudiodev.oss_audio_device method)
  (pipes.Template method)
  (sgmllib.SGMLParser method)
(io.TextIOBase method)
(mimetypes.MimeTypeTypes method)
(mmap.mmap method)
(multifile.MultiFile method)
(ossaudiodev.oss_audio_device method)
(robotparser.RobotFileParser method)
(zipfile.ZipFile method)
read1() (io.BufferedIOBase method)
   (io.BufferedReader method)
   (io.BytesIO method)
read_all() (telnetlib.Telnet method)
read_byte() (mmap.mmap method)
read_eager() (telnetlib.Telnet method)
read_history_file() (in module readline)
read_init_file() (in module readline)
read_lazy() (telnetlib.Telnet method)
read_mime_types() (in module mimetypes)
READ_REstricted
read_sb_data() (telnetlib.Telnet method)
read_some() (telnetlib.Telnet method)
read_token() (shlex.shlex method)
read_until() (telnetlib.Telnet method)
read_very_eager() (telnetlib.Telnet method)
read_very_lazy() (telnetlib.Telnet method)
read_windows_registry() (mimetypes.MimeTypeTypes method)
READABLE (in module Tkinter)
readable() (asyncore.dispatcher method)
   (io.IOBase method)
readall() (io.RawIOBase method)
readbufferproc (C type)
reader() (in module csv)
ReadError
readfp() (ConfigParser.RawConfigParser method)
   (mimetypes.MimeTypeTypes method)
readframes() (aiff.aiff method)
   (sunau.AU_read method)
(read1() (io.BufferedIOBase method)
   (io.BufferedReader method)
   (io.BytesIO method)
read_all() (telnetlib.Telnet method)
read_byte() (mmap.mmap method)
read_eager() (telnetlib.Telnet method)
read_history_file() (in module readline)
read_init_file() (in module readline)
read_lazy() (telnetlib.Telnet method)
read_mime_types() (in module mimetypes)
READ_REstricted
read_sb_data() (telnetlib.Telnet method)
read_some() (telnetlib.Telnet method)
read_token() (shlex.shlex method)
read_until() (telnetlib.Telnet method)
read_very_eager() (telnetlib.Telnet method)
read_very_lazy() (telnetlib.Telnet method)
read_windows Registry() (mimetypes.MimeTypeTypes method)
READABLE (in module Tkinter)
readable() (asyncore.dispatcher method)
   (io.IOBase method)
readall() (io.RawIOBase method)
readbufferproc (C type)
reader() (in module csv)
ReadError
readfp() (ConfigParser.RawConfigParser method)
   (mimetypes.MimeTypeTypes method)
readframes() (aiff.aiff method)
   (sunau.AU_read method)
(wave.Wave_read method)
readinto() (io.BufferedIOBase method)
   (io.RawIOBase method)
readline
module
readline (module)
readline() (bz2.BZ2File method)
   (codecs.StreamReader method)
   (distutils.text_file.TextFile method)
   (file method), [1]
   (imaplib.IMAP4 method)
   (io.IOBase method)
   (io.TextIOBase method)
   (mmap.mmap method)
   (multifile.MultiFile method)
readlines() (bz2.BZ2File method)
   (codecs.StreamReader method)
   (distutils.text_file.TextFile method)
   (file method)
   (io.IOBase method)
   (multifile.MultiFile method)
readlink() (in module os)
readmodule() (in module pyclbr)
readmodule_ex() (in module pyclbr)
READONLY
readonly (memoryview attribute)
readPlist() (in module plistlib)
readPlistFromResource() (in module plistlib)
readPlistFromString() (in module plistlib)
readscaled() (in module imgfile)
READY (in module cd)
ready() (multiprocessing.pool.AsyncResult method)
Real (class in numbers)
real (numbers.Complex attribute)
Real Media File Format
real_quick_ratio() (difflib.SequenceMatcher method)
realloc()
retrbinary() (ftplib method)
retrieve() (urllib.URLopener method)
retrlines() (ftplib method)
return
statement, [1]
return_ok() (cookielib.CookiePolicy method)
RETURN_VALUE (opcode)
returncode (subprocess.CalledProcessError attribute)
(retries attribute)
reverse (collections.deque method)
   (in module audioop)
   (list method)
reverse_order() (collections.deque method)
reversed() (built-in function)
revert() (cookielib.CookieJar method)
rewind() (aifc.aifc method)
   (sunau.AU_read method)
   (wave.Wave_read method)
rewindbody() (rfc822.Message method)
rexec
module
REExec (class in rexec)
RFC
RFC 1014, [1]
RFC 1123
RFC 1321, [1]
RFC 1422
RFC 1521, [1]
RFC 1522
RFC 1524, [1]
RFC 1725
realpath() (in module os.path)
reason (exceptions.UnicodeError attribute)
  (httpplib.HTTPResponse attribute)
  (ssl.SSLError attribute)
  (urllib2.HTTPError attribute)
  (urllib2.URLError attribute)
reattach() (ttk.Treeview method)
rebinding
  name
reccontrols() (ossaudiodev.oss_mixer_device method)
recent() (imaplib.IMAP4 method)
rect() (in module cmath)
rectangle() (in module curses.textpad)
recursive
  object
recv() (asyncore.dispatcher method)
    (multiprocessing.Connection method)
  (socket.socket method)
recv_bytes() (multiprocessing.Connection method)
recv_bytes_into() (multiprocessing.Connection method)
recv_into() (socket.socket method)
recvfrom() (socket.socket method)
recvfrom_into() (socket.socket method)
redirect_request() (urllib2.HTTPRedirectHandler method)
redisplay() (in module readline)
redraw_form() (fl.form method)
redrawln() (curses.window method)
redrawwin() (curses.window method)
reduce (2to3 fixer)
reduce() (built-in function)
  (in module functools)
ref (class in weakref)
reference
  attribute

RFC 1730
RFC 1738
RFC 1750
RFC 1766, [1]
RFC 1808, [1]
RFC 1832, [1]
RFC 1866
RFC 1869, [1]
RFC 1894
RFC 2033
RFC 2045, [1]
RFC 2046, [1]
RFC 2047, [1]
RFC 2060, [1]
RFC 2068
RFC 2104, [1]
RFC 2109, [1]
RFC 2231, [:]
RFC 2342
RFC 2368
RFC 2396, [1]
RFC 2487
RFC 2616, [1]
RFC 2732, [1]
RFC 2774
RFC 2817
RFC 2818
RFC 2821
RFC 2822, [:]
[11], [12], [1;]
[21], [22], [2;]
[31], [32], [33]
RFC 2964
reference count
reference counting
ReferenceError, [1]
ReferenceType (in module weakref)
refilemessages() (mhlib.Folder method)
refresh() (curses.window method)
REG_BINARY (in module _winreg)
REG_DWORD (in module _winreg)
REG_DWORD_BIG_ENDIAN (in module _winreg)
REG_DWORD_LITTLE_ENDIAN (in module _winreg)
REG_EXPAND_SZ (in module _winreg)
REG_FULL_RESOURCE_DESCRIPTOR (in module _winreg)
REG_LINK (in module _winreg)
REG_MULTI_SZ (in module _winreg)
REG_NONE (in module _winreg)
REG_RESOURCE_LIST (in module _winreg)
REG_RESOURCE_REQUIREMENTS_LIST (in module _winreg)
REG_SZ (in module _winreg)
RegexObject (class in re)
register() (abc.ABCMeta method)
    (in module atexit)
    (in module codecs)
    (in module webbrowser)
    (multiprocessing.managers.BaseManager method)
    (select.epoll method)
    (select.poll method)
register_adapter() (in module sqlite3)
register_archive_format() (in module shutil)
register_converter() (in module sqlite3)
register_dialect() (in module csv)
register_error() (in module codecs)
register_function()
(SimpleXMLRPCServer.cgiXMLRPCRequestHandler method)
    (SimpleXMLRPCServer.SimpleXMLRPCServer

RFC 2965, [1]
RFC 3207
RFC 3229
RFC 3280
RFC 3454
RFC 3490, [1]
RFC 3492, [1]
RFC 3493
RFC 3548, [1]
RFC 3986, [1]
RFC 4122, [1]
RFC 4217
RFC 4366
RFC 4627, [1]
RFC 5246
RFC 5280
RFC 5929
RFC 6066
RFC 6125
RFC 7159, [1]
RFC 7301, [1]
RFC 821, [1]
RFC 822, [1], [12], [13], [14]
RFC 854, [1]
RFC 959
RFC 977
rfc2109 (cookielib attribute)
rfc2109_as_nets (cookielib module)
rfc2965 (cookielib module)
rfc822 (module)
rfc822_escape()
method) register_instance()
(SimpleXMLRPCServer.CGIHTTPRequestHandler
method)
(SimpleXMLRPCServer.SimpleXMLRPCServer
method)
register_introspection_functions()
(SimpleXMLRPCServer.CGIHTTPRequestHandler
method)
(SimpleXMLRPCServer.SimpleXMLRPCServer
method)
register_multicall_functions()
(SimpleXMLRPCServer.CGIHTTPRequestHandler
method)
(SimpleXMLRPCServer.SimpleXMLRPCServer
method)
register_namespace() (in module xml.etree.ElementTree)
register_optionflag() (in module doctest)
register_shape() (in module turtle)
registerDOMImplementation() (in module xml.dom)
registerResult() (in module unittest)
relative
  URL
  import
release() (in module platform)
  (logging.Handler method)
  (multiprocessing.Lock method)
  (multiprocessing.RLock method)
  (thread.lock method)
  (threading.Condition method)
  (threading.Lock method)
  (threading.RLock method)
  (threading.Semaphore method)
release_lock() (in module imp)
reload
RFC_4122 (in module
rfile (BaseHTTPServer.attribute)
rfind() (in module
  mmap.mmap
  (str method)
rgb_to_hls() (in module
colorsys)
rgb_to_hsv() (in module
colorsys)
rgb_to_yiq() (in module
colorsys)
RGBColor (class
turtle)
right (filecmp.dircmp)
right() (in module
cmp
dircmp)
right_list (filecmp.dircmp)
right_only (filecmp
cmp)
RIGHTSHIFT (in module
token)
RIGHTSHIFTEQ
rindex() (in module
  (str method)
rjust() (in module
  (str method)
rlcompleter module
rlcompleter (module
rlecode_hqx() (in
rledecode_hqx() (in
RLIM_INFINITY
RLIMIT_AS (in
RLIMIT_CORE (in
RLIMIT_CPU (in
RLIMIT_DATA (in
RLIMIT_FSIZE (in
RLIMIT_MEMLOCK (in
RLIMIT_NOFILE (in
RLIMIT_NPROC (in
RLIMIT_OFILE (in
RLIMIT_RSS (in
RLIMIT_STACK (in
RLIMIT_VMEM (in
built-in function, [1], [2], [3]
reload() (built-in function)
relpath() (in module os.path)
remainder() (decimal.Context method)
remainder_near() (decimal.Context method)
   (decimal.Decimal method)
remove() (array.array method)
   (collections.deque method)
   (in module os)
   (list method)
   (mailbox.MH method)
   (mailbox.Mailbox method)
   (set method)
   (xml.etree.ElementTree.Element method)
remove_flag() (mailbox.MaildirMessage method)
   (mailbox.MMDFMessage method)
   (mailbox.mboxMessage method)
remove_folder() (mailbox.Maildir method)
   (mailbox.MH method)
remove_history_item() (in module readline)
remove_label() (mailbox.BabylMessage method)
remove_option() (ConfigParser.RawConfigParser method)
   (optparse.OptionParser method)
remove_pyc() (msilib.Directory method)
remove_section() (ConfigParser.RawConfigParser method)
remove_sequence() (mailbox.MHMessage method)
remove_tree() (in module distutils.dir_util)
removeAttribute() (xml.dom.Element method)
removeAttributeNode() (xml.dom.Element method)
removeAttributeNS() (xml.dom.Element method)
removeChild() (xml.dom.Node method)
removedirs() (in module os)
removeFilter() (logging.Handler method)
   (logging.Logger method)
removeHandler() (in module unittest)

   RLock (class in multiprocessing)
   RLock() (in module threading)
   rmd() (ftp.ftp)
   rmdir() (in module os)
   RMFF
   rms() (in module audioop)
   rmtree() (in module shutil)
   rnopen() (in module mailbox)
RobotFileParser
robotparser (module)
robots.txt
rollback() (sqlite3)
ROT_FOUR (opcode)
ROT_THREE (opcode)
ROT_TWO (opcode)
rotate() (collections)
   (decimal.Context method)
   (decimal.Decimal method)
RotatingFileHandler
round() (built-in function)
Rounded (class)
Row (class in sqlite3)
row_factory (sqlite3.Connection attribute)
rowcount (sqlite3.Cursor attribute)
RPAR (in module token)
rpartition() (str method)
RSQB (in module token)
rstrip() (in module string)
removemessages() (mhlib.Folder method)
removeResult() (in module unittest)
rename() (ftplib.FTP method)
  (imaplib.IMAP4 method)
  (in module os)
renames (2to3 fixer)
S (in module re)
S_ENFMT (in module stat)
s_eval() (rexec.RExec method)
s_exec() (rexec.RExec method)
s_execfile() (rexec.RExec method)
S_IEXEC (in module stat)
S_IFBLK (in module stat)
S_IFCHR (in module stat)
S_IFDIR (in module stat)
S_IFIFO (in module stat)
S_IFLNK (in module stat)
S_IFMT() (in module stat)
S_IFREG (in module stat)
S_IFSOCK (in module stat)
S_IMODE() (in module stat)
s_import() (rexec.RExec method)
S_IREAD (in module stat)
S_IRGRP (in module stat)
S_IROTH (in module stat)
S_IRUSR (in module stat)
S_IRWXG (in module stat)
S_IRWXO (in module stat)
S_IRWXU (in module stat)
S_ISBLK() (in module stat)
S_ISCHR() (in module stat)
S_ISDIR() (in module stat)
S_ISFIFO() (in module stat)
S_ISGID (in module stat)
S_ISLNK() (in module stat)
S_ISREG() (in module stat)
S_ISSOCK() (in module stat)
S_ISUID (in module stat)
S_ISVTX (in module stat)
S_IWGRP (in module stat)
S_IWOTH (in module stat)
S_IWOTH (in module stat)
S_IREAD (in module stat)
S_IRGRP (in module stat)
S_IROTH (in module stat)
S_IRUSR (in module stat)
S_IWGRP (in module stat)
S_IWOTH (in module stat)
S_IEXEC (in module stat)
S_IFBLK (in module stat)
S_IFCHR (in module stat)
S_IFDIR (in module stat)
S_IFIFO (in module stat)
S_IFLNK (in module stat)
S_IFMT() (in module stat)
S_IFREG (in module stat)
S_IFSOCK (in module stat)
S_IMODE() (in module stat)
simport() (rexec.RExec method)
showsyntaxerror() (code.InteractiveInterpreter method)
showtraceback() (code.InteractiveInterpreter method)
showturtle() (in module turtle)
showwarning() (in module warnings)
shuffle() (in module random)
shutdown() (in module imaplib)
socketserver.BaseServer method
findertools module
logging module
multiprocessing.managers.BaseManager method
socket.socket method
shutil (module)
SIG_DFL (in module signal)
SIG_IGN (in module signal)
SIGINT, [1]
siginterrupt() (in signal module, [1]
signal (module)
signal() (in module)
simple
statement
Simple Mail Transfer Protocol
SimpleCookie (class in Cookie)
simplefilter() (in module warnings)
SimpleHandler
SimpleHTTPRe
SimpleHTTPSe
SimpleHTTPSe module
SimpleHTTPSe
S_IWRITE (in module stat)
S_IWUSR (in module stat)
S_IXGRP (in module stat)
S_IXOTH (in module stat)
S_IXUSR (in module stat)
s_reload() (rexec.RExec method)
s_unload() (rexec.RExec method)
safe_substitute() (string.Template method)
SafeConfigParser (class in ConfigParser)
saferepr() (in module pprint)
same_files (filecmp.dircmp attribute)
same_quantum() (decimal.Context method)
    (decimal.Decimal method)
samefile() (in module os.path)
sameopenfile() (in module os.path)
samestat() (in module os.path)
sample() (in module random)
save() (cookielib.FileCookieJar method)
save_bgn() (htmllib.HTMLParser method)
save_end() (htmllib.HTMLParser method)
SaveKey() (in module _winreg)
savetty() (in module curses)
SAX2DOM (class in xml.dom.pulldom)
SAXException
SAXNotRecognizedException
SAXNotSupportedException
SAXParseException
scale() (in module imageop)
scaleb() (decimal.Context method)
    (decimal.Decimal method)
scalebarvalues() (FrameWork.ScrolledWindow method)
scanf()
sched (module)
scheduler (class in sched)
schema (in module msilib)
sci() (in module fpformat)
scope, [1]

SimpleXMLRPCClient
SimpleXMLRPCServer
SimpleXMLRPCServer
SimpleXMLRPCServer
(module)

sin() (in module math)
    (in module re)
singleton
tuple

sinh() (in module math)
    (in module re)
site (module)
site command line
    --user-base
    --user-site

site-packages
directory

site-python
directory

sitecustomize
module

size (struct.Struct method)
    (tarfile.TarInfo method)

size() (ftpplib.FTP method)
    (mmap.mmap method)

Sized (class in functools)

sizeof() (in module struct)

SKIP (in module unittest)

skip() (chunk.Chunk method)
    (in module unittest)

skipf() (in module unittest)

skipinitialspace (csv.Dialect attribute)

skipped (unittest.TestResult attribute)

skippedEntity() (xml.sax.handler.ContentHandler method)

SkipTest

skipTest() (unittest method)
Scrap Manager
Screen (class in turtle)
screensize() (in module turtle)
script_from_examples() (in module doctest)
scroll() (curses.window method)
scrollbar_callback() (FrameWork.ScrolledWindow method)
scrollbars() (FrameWork.ScrolledWindow method)
ScrolledCanvas (class in turtle)
ScrolledText (module)
scrolllok() (curses.window method)
search
  path, module, [1], [2], [3], [4], [5], [6]
search() (imaplib.IMAP4 method)
    (in module re)
      (re.RegexObject method)
SEARCH_ERROR (in module imp)
second (datetime.datetime attribute)
    (datetime.time attribute)
section_divider() (multifile.MultiFile method)
sections() (ConfigParser.RawConfigParser method)
secure (cookielib.Cookie attribute)
Secure Hash Algorithm
secure hash algorithm, SHA1, SHA224, SHA256, SHA384, SHA512
Secure Sockets Layer
security
  CGI
see() (ttk.Treeview method)
seed() (in module random)
seek() (bz2.BZ2File method)
  (chunk.Chunk method)
    (file method)
      (io.IOBase method)
        (io.TextIOBase method)
          (mmap.mmap method)
            (multifile.MultiFile method)
SEEK_CUR (in module os)
    (in module posixfile)
SEEK_END (in module os)
    (in module posixfile)
SEEK_SET (in module os)
    (in module posixfile)
seekable() (io.IOBase method)
segcountproc (C type)
Select (class in Tix)
select (module)
select() (imaplib.IMAP4 method)
    (in module gl)
    (in module select)
    (ttk.Notebook method)
selected_alpn_protocol() (ssl.SSLSocket method)
selected_npn_protocol() (ssl.SSLSocket method)
selection() (ttk.Treeview method)
selection_add() (ttk.Treeview method)
selection_remove() (ttk.Treeview method)
selection_set() (ttk.Treeview method)
selection_toggle() (ttk.Treeview method)
Semaphore (class in multiprocessing)
    (class in threading)
Semaphore()
    (multiprocessing.managers.SyncManager method)
semaphores, binary
SEMI (in module token)
send() (aetools.TalkTo method)
    (asyncore.dispatcher method)
    (generator method)
    (httplib.HTTPConnection method)
    (imaplib.IMAP4 method)
    (logging.handlers.DatagramHandler method)
    (logging.handlers.SocketHandler method)
    (multiprocessing.Socket method)
    (multiprocessing.Connection method)
send_bytes() (multiprocessing.Connection method)
SND_ALIAS (in module winsound)
SND_ASYNC (in module winsound)
SND_FILENAME (in module winsound)
SND_LOOP (in module winsound)
SND_MEMO
SND_NODEFA
SND_NOSTOP
SND_NOWAIT
SND_PURGE
SND_PURGET
sndhdr (module)
sniff() (csv.Sniff
Sniffer (class in
SOCK_DGRAM
SOCK_RAW (ir
SOCK_RDM (ir
SOCK_SEQPA
SOCK_STREA
socket
    module, [1]
    object
socket (module
    (SocketServ
    socket() (imapli
    (in module socket
    socket_type (SocketServ
    SocketHandler
    socketpair() (in
    SocketServer (in
    SocketType (in
    softspace (file
    SOMAXCONN
    sort() (imaplib.I
    (list method,
    sort_stats() (psl
    sorted() (built-in
    sortTestMethod
    attribute)
send_error()
(BaseHTTPServer.BaseHTTPRequestHandler method)
send_flowing_data() (formatter.writer method)
send_header()
(BaseHTTPServer.BaseHTTPRequestHandler method)
send_hor_rule() (formatter.writer method)
send_label_data() (formatter.writer method)
send_line_break() (formatter.writer method)
send_literal_data() (formatter.writer method)
send_paragraph() (formatter.writer method)
send_response()
(BaseHTTPServer.BaseHTTPRequestHandler method)
send_signal() (subprocess.Popen method)
sendall() (socket.socket method)
sendcmd() (ftplib.FTP method)
sendfile() (wsgiref.handlers.BaseHandler method)
sendmail() (smtplib.SMTP method)
sendto() (socket.socket method)
sep (in module os)
Separator() (in module FrameWork)

sequence
  item
  iteration
    object, [1], [2], [3], [4], [5], [6], [7], [8]
  types, mutable
    types, operations on, [1]
Sequence (class in collections)
sequence (in module msilib)
sequence2st() (in module parser)
sequenceIncludes() (in module operator)
SequenceMatcher (class in difflib), [1]
SerialCookie (class in Cookie)

serializing
  objects
serve_forever() (SocketServer.BaseServer method)

source (doctest)
  (shlex.shlex source_character sourcehook() (space
  span() (re.Match spawn() (distute
    (in module p
  spawnl() (in m
  spawnlp() (in m
  spawnlpe() (in r
  spawnv() (in m
  spawnve() (in m
  spawnvpe() (in

special
  attribute
  attribute, ge
  specified_attributes
  (xml.parsers.expat.xmlparser speed() (in modu
  (ossaudiodev.
splash() (in module
  split() (in modu
  (in module r
  (in module s
  (in module s
  (re.RegexOl
  (str method)
split_quoted() (in
  splitdrive() (in m
  splitext() (in mo
  splitfields() (in r
  splitlines() (str r
server

    WWW, [1]
server (BaseHTTPServer.BaseHTTPRequestHandler attribute)
server_activate() (SocketServer.BaseServer method)
server_address (SocketServer.BaseServer attribute)
server_bind() (SocketServer.BaseServer method)
server_close() (SocketServer.BaseServer method)
server_software (wsgiref.handlers.BaseHandler attribute)
server_version
    (BaseHTTPServer.BaseHTTPRequestHandler attribute)
    (SimpleHTTPServer.SimpleHTTPRequestHandler attribute)
ServerProxy (class in xmlrpclib)
session_stats() (ssl.SSLContext method)
set
    display
    object, [1], [2], [3]
set (built-in class)
Set (class in collections)
    (class in sets)
Set Breakpoint
set type
    object
set() (ConfigParser.RawConfigParser method)
    (ConfigParser.SafeConfigParser method)
    (Cookie.Morsel method)
    (EasyDialogs.ProgressBar method)
    (ossaudiodev.oss_mixer_device method)
    (test.test_support.EnvironmentVarGuard method)
    (threading.Event method)
    (ttk.Comboobox method)
    (ttk.Treeview method)
    (xml.etree.ElementTree.Element method)
set_all()
set_allowed_domains() (cookielib.DefaultCookiePolicy method)
set_alpn_protocols() (ssl.SSLContext method)
set_app() (wsgiref.simple_server.WSGIServer method)
set_authorizer() (sqlite3.Connection method)
set_blocked_domains() (cookielib.DefaultCookiePolicy method)
set_boundary() (email.message.Message method)
set_break() (bdb.Bdb method)
set_charset() (email.message.Message method)
set_children() (ttk.Treeview method)
set_ciphers() (ssl.SSLContext method)
set_completer() (in module readline)
set_completer_delims() (in module readline)
set_completion_display_matches_hook() (in module readline)
set_continue() (bdb.Bdb method)
set_conversion_mode() (in module ctypes)
set_cookie() (cookielib.CookieJar method)
set_cookie_if_ok() (cookielib.CookieJar method)
set_current() (msilib.Feature method)
set_date() (mailbox.MaildirMessage method)
set_debug() (in module gc)
set_debuglevel() (ftplib.FTP method)
   (httplib.HTTPConnection method)
   (ntplib.NNTP method)
   (poplibPOP3 method)
   (smtplib.SMTP method)
   (telnetlib.Telnet method)
set_default_type() (email.message.Message method)
set_default_verify_paths() (ssl.SSLContext method)
set_defaults() (argparse.ArgumentParser method)
   (optparse.OptionParser method)
set_ecdh_curve() (ssl.SSLContext method)
set_errno() (in module ctypes)
set_event_call_back() (in module fl)
set_executable() (in module multiprocessing)
set_executables() (distutils.ccompiler.CCompiler method)
set_ecdh_curve() (ssl.SSLContext method)
set_flags() (mailbox.MaildirMessage method)  
  (mailbox.MMDFMessage method)  
  (mailbox.mboxMessage method)  
set_form_position() (fl.form method)  
set_from() (mailbox.mboxMessage method)  
  (mailbox.MMDFMessage method)  
set_graphics_mode() (in module fl)  
set_history_length() (in module readline)  
set_include_dirs() (distutils.ccompiler.CCompiler method)  
set_info() (mailbox.MaildirMessage method)  
set_labels() (mailbox.BabylMessage method)  
set_last_error() (in module ctypes)  
set_libraries() (distutils.ccompiler.CCompiler method)  
set_library_dirs() (distutils.ccompiler.CCompiler method)  
SET_LINENO (opcode)  
set_link_objects() (distutils.ccompiler.CCompiler method)  
set literal (2to3 fixer)  
set_location() (bsddb.bsddbobject method)  
set_next() (bdb.Bdb method)  
set_nonstandard_attr() (cookielib.Cookie method)  
set_npn_protocols() (ssl.SSLContext method)  
set_ok() (cookielib.CookiePolicy method)  
set_option_negotiation_callback() (telnetlib.Telnet method)  
set_position() (xdrlib.Unpacker method)  
set_pre_input_hook() (in module readline)  
set_progress_handler() (sqlite3.Connection method)  
set_proxy() (urllib2.Request method)  
set_python_build() (in module distutils.sysconfig)  

StartCdataSectionHandler() (xml.parsers.expat.xmlparser method)  
StartDoctypeDeclHandler() (xml.parsers.expat.xmlparser method)  
startDocument() (xml.sax.handler.ContentHandler method)  
StartElement() (xml.sax.handler.ContentHandler method)  
StartElementHandler() (xmlparsers.expat.xmlparser method)  
StartElementNS() (xml.sax.handler.ContentHandler method)  
startfile() (in module os)  
startmultipartbody() (MimeWriter.MimeWriter method)  
StartNamespaceDeclHandler() (xml.parsers.expat.xmlparser method)  
startPrefixMapping() (xml.sax.handler.ContentHandler method)  
startswith() (str method)  
startTest() (unittest.TestResult method)  
startTestRun() (unittest.TestResult method)  
starttls() (smtplib.SMTP method)  
STARTF_USESHOWWINDOW (in module subprocess)  
STARTF_USESTDHANDLES (in module subprocess)  
stat() (in module os)  
stat_float_times() (in module os)  
state() (ttk.Widget method)  
statvfs() (nntplib.NNTP method)  
statvfs() (poplib.POP3 method)  
statement
  * [1]
set_quit() (bdb.Bdb method)
set_recsrc() (ossaudiodev.oss.mixer_device method)
set_return() (bdb.Bdb method)
set_runtime_library_dirs()
  (distutils.ccompiler.CCompiler method)
set_seq1() (difflib.SequenceMatcher method)
set_seq2() (difflib.SequenceMatcher method)
set_seqs() (difflib.SequenceMatcher method)
set_sequences() (mailbox.MH method)
  (mailbox.MHMessage method)
set_server_documentation()
  (DocXMLRPCServer.DocCGIXMLRPCRequestHandler method)
  (DocXMLRPCServer.DocXMLRPCServer method)
set_server_name()
  (DocXMLRPCServer.DocCGIXMLRPCRequestHandler method)
  (DocXMLRPCServer.DocXMLRPCServer method)
set_server_title()
  (DocXMLRPCServer.DocCGIXMLRPCRequestHandler method)
  (DocXMLRPCServer.DocXMLRPCServer method)
set_servername_callback() (ssl.SSLContext method)
set_spacing() (formatter.formatter method)
set_startup_hook() (in module readline)
set_step() (bdb.Bdb method)
set_subdir() (mailbox.MaildirMessage method)
set_terminator() (asynchat.async_chat method)
set_threshold() (in module gc)
set_trace() (bdb.Bdb method)
  (in module bdb)
  (in module pdb)
  (pdb.Pdb method)
set_tunnel() (httpplib.HTTPConnection method)
set_type() (email.message.Message method)
set_unittest_reportflags() (in module doctest)
set_unixfrom() (email.message.Message method)
set_until() (bdb.Bdb method)

**, [1] @
assert, [1]
assignment,
assignment,
break, [1], [2]
class
compound
continue, [1]
def
del, [1], [2],
except
exec, [1], [2]
expression
for, [1], [2], [3]
from
future
global, [1], [2]
if, [1]
import, [1], [2]
loop, [1], [2]
raise, [1]
return, [1], [2]
simple
try, [1], [2]
while, [1], [2]
with, [1]
yield
statement group

staticmethod
built-in funct
staticmethod() (Stats (class in p...
set_url() (robotparser.RobotFileParser method)
set_usage() (optparse.OptionParser method)
set_userptr() (curses.panel.Panel method)

set_visible() (mailbox.BabylMessage method)
set_wakeup_fd() (in module signal)
setacl() (imaplib.IMAP4 method)
setannotation() (imaplib.IMAP4 method)
setarrowcursor() (in module FrameWork)

set_visible() (built-in function)

setAttribute() (xml.dom.Element method)
setAttributeNode() (xml.dom.Element method)
setAttributeNodeNS() (xml.dom.Element method)
SetBase() (xml.parsers.expax.xmlparser method)
setblocking() (socket.socket method)
setByteStream() (xml.sax.xmlreader.InputSource method)

setcbreak() (in module tty)
setcharacterstream() (xml.sax.xmlreader.InputSource method)

setcheckinterval() (in module sys), [1]

setcomptype() (aifc.aifc method)

setdefault() (dict method)

setdefaultencoding() (in module sys)

setdefaulttimeout() (in module socket)

setdlopenflags() (in module sys)

setDocumentLocator() (xml.sax.handler.ContentHandler method)

setDTDHandler() (xml.sax.xmlreader.XMLReader method)

setcontext() (in module decimal)

SetCreatorAndType() (in module MacOS)

setCurrent() (mhib.Folder method)

setDaemon() (threading.Thread method)

setdefault() (dict method)

setdefaultencoding() (in module sys)

setdefaulttimeout() (in module socket)

setdlopenflags() (in module sys)

setDocumentLocator()

setStatus() (httplib.HTTPResponse attribute)

setstatus() (imaplib.IMAP4 method)

statvfs() (module)

statvfs() (in module subprocess)

STD_ERROR_HANDLE (in module subprocess)

STD_INPUT_HANDLE (in module subprocess)

STD_OUTPUT_HANDLE (in module subprocess)

StdButtonBox (method)

stderr (in module sys)

stdout (in module subprocess)

Stein, Greg

step() (slice object)

step() (tkk.Progressbar method)

stereocontrols() (ossaudiodev.oss_mixer_device method)

STILL (in module cd)

stop() (slice object)

stop() (hotshot.Profile method)

(unicode.TestResult method)

STOP_CODE (opcode)

stop_here() (bdb.Bdb method)

StopIteration

exception, [:

stopListening() (unittest.TestResult method)

stopTest() (unittest.TestResult method)

stopTestRun() (unittest.TestResult method)
method)
setegid() (in module os)
setEncoding() (xml.sax.xmlreader.InputSource
method)
setEntityResolver() (xml.sax.xmlreader.XMLReader
method)
setErrorHandler() (xml.sax.xmlreader.XMLReader
method)
seteuid() (in module os)
setFeature() (xml.sax.xmlreader.XMLReader method)
setfirstweekday() (in module calendar)
setfmt() (ossaudiodev.oss_audio_device method)
setFormatter() (logging.Handler method)
setframerate() (aifc.aifc method)
  (sunau.AU_write method)
  (wave.Wave_write method)
setgid() (in module os)
setgroups() (in module os)
seth() (in module turtle)
setheading() (in module turtle)
SetInteger() (msilib.Record method)
setItem() (in module operator)
setitimer() (in module signal)
setlast() (mhlib.Folder method)
setLevel() (logging.Handler method)
  (logging.Logger method)
setlocale() (sgmllib.SGMLParser method)
setlocale() (in module locale)
setLocale() (xml.sax.xmlreader.XMLReader method)
setLoggerClass() (in module logging)
setlogmask() (in module syslog)
setmark() (aifc.aifc method)
setMaxConns() (urllib2.CacheFsocket method)
setmode() (in module msvcr
setName() (threading.Thread method)
setnchannels() (aifc.aifc method)
  (sunau.AU_write method)
  (wave.Wave_write method)
storbinary() (ftp
store() (imaplib.
STORE_ACTIC
STORE_ATTR
STORE_DEREN
STORE_FAST
STORE_GLOB
STORE_GLOBAL
STORE_NAME
STORE_SLICE
STORE_SLICE
STORE_SLICE
STORE_SLICE
STORE_SUBS
storlines() (ftplib:
str
  built-in funct
  format
str (built-in class:
str() (in module
strcoll() (in mod
StreamError
StreamHandler
StreamReader
StreamReader
StreamWriter (c
strerror() (in mod
strftime() (datetime.date
  (datetime.datetime
  (datetime.time
  (in module
time)
strict (csv.Dialect)
setnframes() (aifc.aifc method)  
(sunau.AU_write method)  
(wave.Wave_write method)  
(setnomoretags() (sgmllib.SGMLParser method)
setoption() (in module jpeg)
SetParamEntityParsing() (xml.parsers.expat.xmlparser method)
setparameters() (ossaudiodev.oss_audio_device method)
setparams() (aifc.aifc method)  
(in module al)  
(sunau.AU_write method)  
(wave.Wave_write method)  
setpassword() (zipfile.ZipFile method)
setpath() (in module fm)
setpgid() (in module os)
setpgrp() (in module os)
setpos() (aifc.aifc method)  
(in module turtle)  
(sunau.AU_read method)  
(wave.Wave_read method)  
setposition() (in module turtle)
setprofile() (in module sys)  
(in module threading)
SetProperty() (msilib.SummaryInformation method)
setProperty() (xml.sax.xmlreader.XMLReader method)
setPublicId() (xml.sax.xmlreader.InputSource method)
setquota() (imaplib.IMAP4 method)
setrain() (in module tty)
setrecursionlimit() (in module sys)
setregid() (in module os)
setresgid() (in module os)
setresuid() (in module os)
setreuid() (in module os)
setrlimit() (in module resource)
sets (module)
setsampwidth() (aifc.aifc method)  
(strict_domain (attribute)
strict_errors() (in module http.cookies)
strict_ns_domain (cookielib.DefaultCookiePolicy attribute)
strict_ns_set_initial_dollar (cookielib.DefaultCookiePolicy attribute)
strict_ns_set_path (cookielib.DefaultCookiePolicy attribute)
strict_ns_unverifiable (cookielib.DefaultCookiePolicy attribute)
strict_rfc2965_unverifiable (cookielib.DefaultCookiePolicy attribute)
strides (memoryview attribute)
string
  Unicode
  comparison
  conversion,
  formatting
  interpolation
  item
  methods
  module, [1],
  object, [1], [2]
STRING (in module token)
string (module)
(re.MatchObject attribute)
string literal
string_at() (in module cffi
StringIO (class
(StringIO (class in StringIO
(module)
StringIO() (in module cStringIO
StringTypes (in module types
strings, documentation
StringType (in module types)
setvbuf()
setwatchcursor() (in module FrameWork)
setworldcoordinates() (in module turtle)
setx() (in module turtle)
sety() (in module turtle)
SF_APPEND (in module stat)
SF_ARCHIVED (in module stat)
SF_IMMUTABLE (in module stat)
SF_NOUNLINK (in module stat)
SF_SNAPSHOT (in module stat)
SGML
sgmllib
    module
sgmllib (module)
SGMLParseError
SGMLParser (class in sgmllib)
    (in module sgmllib)
sha (module)
Shape (class in turtle)
shape (memoryview attribute)
shape() (in module turtle)
shapesize() (in module turtle)
shared_object_filename()
    (distutils.ccompiler.CCompiler method)
Shelf (class in shelve)
shelve
    module
shelve (module)
shift() (decimal.Context method)
    (decimal.Decimal method)
shift_path_info() (in module wsgiref.util)
shifting
    operation
    operations
shlex (class in shlex)
    (module)
shortDescription() (unittest.TestCase method)
shouldFlush() (logging.handlers.BufferingHandler
subprocess (module)
subscribe() (imaplib.IMAP4 method)
subsript
    assignment
    operation
subscription, [1
    assignment
    subsequent_indent
    attribute
substitution (in re
subwin() (in curses
successful()
    (multiprocessing
suffix_map (in r
    (mimetypes.
suite
suite() (in module
suiteClass (unittest
sum() (built-in function)
sum_list()
sum_sequence
summarize() (doctest.
sunau (module)
    SUNAUDIODEV
    module
sunaudiodev
    module
SUNAUDIODEV
sunaudiodev
    super (pyclbr.
    super() (built-in
    supports_unicode_filenames
    (os.path)
method)
    (logging.handlers.MemoryHandler method)
shouldStop (unittest.TestResult attribute)
show() (curses.panel.Panel method)
show_choice() (in module fl)
show_compilers() (in module distutils.ccompiler)
show_file_selector() (in module fl)
show_form() (fl.form method)
show_input() (in module fl)
show_message() (in module fl)
show_question() (in module fl)
suppression
    newline
    SW_HIDE (in module subprocess)
swapcase() (in module string)
sym() (dl.dl method)
sym_name (in module Symbol)
Symbol (class in module SymbolTable)
SymbolTable (class in symtable)
symlink() (in module os)
symmetric_difference() (set method)
symmetric_difference_update() (set method)
symtable (module)
symtable() (in module symtable)
sync() (bsddb.bdbhash method)
syncdown() (in module curses)
synchronized() (in module sys)
SyncManager (in module multiprocessing)
syncok() (in module shelve)
syncup() (in module curses)
syntax, [1]
SyntaxErr
SyntaxError
SyntaxWarning
sys
    module, [1], sys (module)
sys.exc_info
sys.exc_traceback
sys.last_traceback
sys.meta_path
sys.modules
sys.path
sys.path_hooks
sys.path_importer_cache
sys.stderr
sys.stdin
sys.stdout
sys_exc (2to3 fixer)
sys_version
(BaseHTTPServer.BaseHTTPRequestHandler attribute)
SysBeep() (in module MacOS)
sysconf() (in module os)
sysconf_names
sysconfig (module)
syslog (module)
syslog() (in module syslog)
SysLogHandler (class in logging.handlers)
system() (in module os)
    (in module platform)
system_alias() (in module platform)
SystemError (built-in exception)
SystemExit (built-in exception)
    (built-in exception)
systemId (xml.dom.Document attribute)
SystemRandom (class in random)
SystemRoot
Table

<table>
<thead>
<tr>
<th>Function/Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_FMT (in module locale)</td>
<td>Table formatting in locale.</td>
</tr>
<tr>
<td>T_FMT_AMPM (in module locale)</td>
<td>Table formatting with AM/PM.</td>
</tr>
<tr>
<td>tab</td>
<td>Tab method.</td>
</tr>
<tr>
<td>tab() (ttk.Notebook method)</td>
<td>Tab method.</td>
</tr>
<tr>
<td>TabError</td>
<td>Tab error.</td>
</tr>
<tr>
<td>tabnanny (module)</td>
<td>Tab nanny module.</td>
</tr>
<tr>
<td>tabular</td>
<td>Tabular data.</td>
</tr>
<tr>
<td>data</td>
<td>Tabular data.</td>
</tr>
<tr>
<td>tag (xml.etree.ElementTree.Element attribute)</td>
<td>Tag method.</td>
</tr>
<tr>
<td>tag_bind() (ttk.Treeview method)</td>
<td>Tag bind method.</td>
</tr>
<tr>
<td>tag_configure() (ttk.Treeview method)</td>
<td>Tag configure method.</td>
</tr>
<tr>
<td>tag_has() (ttk.Treeview method)</td>
<td>Tag has method.</td>
</tr>
<tr>
<td>tagName (xml.dom.Element attribute)</td>
<td>Tag name.</td>
</tr>
<tr>
<td>takewhile() (in module itertools)</td>
<td>Take while method.</td>
</tr>
<tr>
<td>TalkTo (class in aetools)</td>
<td>TalkTo method.</td>
</tr>
<tr>
<td>tan() (in module cmath)</td>
<td>Tan method.</td>
</tr>
<tr>
<td>(in module math)</td>
<td>Tan in math.</td>
</tr>
<tr>
<td>tanh() (in module cmath)</td>
<td>Tanh method.</td>
</tr>
<tr>
<td>(in module math)</td>
<td>Tanh in math.</td>
</tr>
<tr>
<td>TarError</td>
<td>Tar error.</td>
</tr>
<tr>
<td>tarfile (module)</td>
<td>Tarfile module.</td>
</tr>
<tr>
<td>TarFileCompat (class in tarfile)</td>
<td>TarfileCompat class.</td>
</tr>
<tr>
<td>TarFileCompat.TAR_GZIPPED (in module tarfile)</td>
<td>TarfileCompat.TAR_GZIPPED.</td>
</tr>
<tr>
<td>TarFileCompat.TAR_PLAIN (in module tarfile)</td>
<td>TarfileCompat.TAR_PLAIN.</td>
</tr>
<tr>
<td>target</td>
<td>Target.</td>
</tr>
<tr>
<td>deletion</td>
<td>Deletion.</td>
</tr>
<tr>
<td>list, [1]</td>
<td>List.</td>
</tr>
<tr>
<td>tix_cget() (Tix.tixCommand method)</td>
<td>Tix cget method.</td>
</tr>
<tr>
<td>tix_configure()</td>
<td>Tix configure method.</td>
</tr>
<tr>
<td>(Tix.tixCommand method)</td>
<td>Tix configure (method).</td>
</tr>
<tr>
<td>tix_filedialog()</td>
<td>Tix filedialog method.</td>
</tr>
<tr>
<td>(Tix.tixCommand method)</td>
<td>Tix filedialog (method).</td>
</tr>
<tr>
<td>tix_getbitmap()</td>
<td>Tix getbitmap method.</td>
</tr>
<tr>
<td>(Tix.tixCommand method)</td>
<td>Tix getbitmap (method).</td>
</tr>
<tr>
<td>tix_getimage()</td>
<td>Tix getimage method.</td>
</tr>
<tr>
<td>(Tix.tixCommand method)</td>
<td>Tix getimage (method).</td>
</tr>
<tr>
<td>TIX_LIBRARY</td>
<td>TIX_LIBRARY.</td>
</tr>
<tr>
<td>tix_option_get()</td>
<td>Tix option get method.</td>
</tr>
<tr>
<td>(Tix.tixCommand method)</td>
<td>Tix option get (method).</td>
</tr>
<tr>
<td>tix_resetoptions()</td>
<td>Tix reset options method.</td>
</tr>
<tr>
<td>(Tix.tixCommand method)</td>
<td>Tix reset options (method).</td>
</tr>
<tr>
<td>tixCommand (class in Tix)</td>
<td>Tix command class.</td>
</tr>
<tr>
<td>Tk</td>
<td>Tk.</td>
</tr>
<tr>
<td>(class in Tkinter)</td>
<td>Tk class in Tkinter.</td>
</tr>
<tr>
<td>Tk Option Data Types</td>
<td>Tk option data types.</td>
</tr>
<tr>
<td>TK_LIBRARY</td>
<td>TK_LIBRARY.</td>
</tr>
<tr>
<td>Tkinter</td>
<td>Tkinter.</td>
</tr>
<tr>
<td>(module)</td>
<td>Tkinter (module).</td>
</tr>
<tr>
<td>TList (class in Tix)</td>
<td>TList class in Tix.</td>
</tr>
<tr>
<td>TLS</td>
<td>TLS.</td>
</tr>
<tr>
<td>TMP, [1]</td>
<td>TMP.</td>
</tr>
<tr>
<td>TMP_MAX (in module os)</td>
<td>TMP_MAX (in os module).</td>
</tr>
<tr>
<td>TMPDIR, [1]</td>
<td>TMPDIR.</td>
</tr>
<tr>
<td>tmpfile() (in module os)</td>
<td>Tmpfile method.</td>
</tr>
<tr>
<td>(in module os)</td>
<td>(in module os).</td>
</tr>
<tr>
<td>tmpnam() (in module os)</td>
<td>Tmpnam method.</td>
</tr>
<tr>
<td>(in module os)</td>
<td>(in module os).</td>
</tr>
<tr>
<td>to_eng_string()</td>
<td>To eng string method.</td>
</tr>
<tr>
<td>to_integral() (decimal.Decimal method)</td>
<td>To integral method.</td>
</tr>
<tr>
<td>to_integral_exact()</td>
<td>To integral exact method.</td>
</tr>
</tbody>
</table>
list assignment
list, deletion
loop control
target (xml.dom.ProcessingInstruction attribute)
TarInfo (class in tarfile)
task_done()
(multiprocessing.JoinableQueue method)
(Queue.Queue method)
tb_frame (traceback attribute)
tb_lasti (traceback attribute)
tb_lineno (module traceback)
tb_lineno() (module traceback)
tb_next (traceback attribute)
tcdrain() (module termios)
tcflow() (module termios)
tcflush() (module termios)
tcgetattr() (module termios)
tcgetATTR() (module termios)
tcgetpgrp() (module os)
Tcl() (module Tkinter)
TCL_LIBRARY
TCPServer (class in SocketServer)
tcsendbreak() (module termios)
tcsetATTR() (module termios)
tcsetpgrp() (module os)
tearDown() (unittest.TestCase method)
tearDownClass() (unittest.TestCase method)
te() (module itertools)
tell() (aifc.aifc method), [1]
(bz2.BZ2File method)
(chunk.Chunk method)
(file method)
(io.IOBase method)
(io.TextIOBase method)
(mmap.mmap method)
(multipile.MultiFile method)
(decimal.Context method)
(decimal.Decimal method)
to_integral_value()
(decimal.Decimal method)
to_sci_string()
(decimal.Context method)
to_splittable()
(email.charset.Charset method)
ToASCII() (module encodings.idna)
tobuf() (tarfile.TarInfo method)
tobytes() (memoryview method)
tochild (popen2.Popen3 attribute)
today() (datetime.date class method)
(datetime.datetime class method)
tofile() (array.array method)
tok_name (module token)
token
(module)
(shlex.shlex attribute)
tokeneater() (module tabnanny)
TokenError
tokenize (module)
tokenize() (module tokenizer)
tolist() (array.array method)
(memoryview method)
(parser.ST method)
tomono() (module audioo)
toordinal() (datetime.date method)
(datetime.datetime method)
(sunau.AU_read method)
(sunau.AU_write method)
(wave.Wave_read method)
(wave.Wave_write method)
Telnet (class in telnetlib)
telnetlib (module)
TEMP
tempdir (in module tempfile)
tempfile (module)
Template (class in pipes)
    (class in string)
template (in module tempfile)
    (string.Template attribute)
tempnam() (in module os)
temporary
    file
    file name
TemporaryFile() (in module tempfile)
TERM, [1]
termattrs() (in module curses)
terminate()
    (multiprocessing.pool.multiprocessing.Pool method)
    (multiprocessing.Process method)
    (subprocess.Popen method)
termination model
termios (module)
termname() (in module curses)
ternary
    operator
test
    identity
    membership
test (doctest.DocTestFailure attribute)
    (doctest.UnexpectedException attribute)
    (module)
method)
top() (curses.panel.Panel method)
    (poplib.POPO3 method)
top_panel() (in module
curses.panel)
toprettyxml()
    (xml.dom.minidom.Node method)
tostereo() (in module audioop)
tostring() (array.array method)
    (in module xml.etree.ElementTree)
tostringlist() (in module
xml.etree.ElementTree)
total_changes
    (sqlite3.Connection attribute)
total_ordering() (in module
functools)
total_seconds()
    (datetime.timedelta method)
totuple() (parser.ST method)
touched() (in module
tacostools)
touchline() (curses.window
method)
touchwin() (curses.window
method)
tounicode() (array.array
method)
ToUnicode() (in module
encodings.idna)
tovideo() (in module imageop)
towards() (in module turtle)
toxml()
    (xml.dom.minidom.Node
method)
.tp_as_mapping (C member,'
test() (in module cgi)
    (mutex.mutex method)
test.test_support (module)
testandset() (mutex.mutex method)
TestCase (class in unittest)
TestFailed
testfile() (in module doctest)
TESTFN (in module test.test_support)
TestLoader (class in unittest)
testMethodPrefix (unittest.TestLoader attribute)
testmod() (in module doctest)
TestResult (class in unittest)
tests (in module imghdr)
testsSource() (in module doctest)
testsRun (unittest.TestResult attribute)
TestSuite (class in unittest)
testzip() (zipfile.ZipFile method)
text (in module msilib)
    (xml.etree.ElementTree.Element attribute)
text() (msilib.Dialog method)
text_factory (sqlite3.Connection attribute)
Textbox (class in curses.textpad)
TextCalendar (class in calendar)
textdomain() (in module gettext)
TextFile (class in distutils.text_file)
TextIOWrapper (class in io)
TextIOWrapper (class in io)
TextTestResult (class in unittest)
TextTestRunner (class in unittest)
textwrap (module)
TextWrapper (class in textwrap)

theme_create() (ttk.Style method)
theme_names() (ttk.Style method)
theme_settings() (ttk.Style method)
theme_use() (ttk.Style method)
THOUSEP (in module locale)

tp_as_number (C member)

    tp_as_sequence (C member)

tparm() (in module curses)

    trace
        stack
Trace (class in trace)
trace (module)
trace command line option
    --help
    --ignore-dir=<dir>
    --ignore-module=<mod>
    --version
    -C, --coverdir=<dir>
    -R, --no-report
    -T, --trackcalls
    -c, --count
    -f, --file=<file>
    -g, --timing
    -l, --listfuncs
    -m, --missing
    -r, --report
    -s, --summary
    -t, --trace

trace function, [1], [2]
trace() (in module inspect)
trace_dispatch() (bdb.Bdb method)

traceback
    object, [1], [2], [3], [4]
traceback (module)
traceback_limit
    (wsgiref.handlers.BaseHandler attribute)
tracebacklimit (in module sys)

tracebacks
    in CGI scripts
thread
    module
Thread (class in threading)
thread (module)
thread() (imaplib.IMAP4 method)
ThreadError
threading (module)
ThreadingMixIn (class in SocketServer)
ThreadingTCPServer (class in SocketServer)
ThreadingUDPServer (class in SocketServer)
threads
  IRIX
  POSIX
throw (2to3 fixer)
throw() (generator method)
tie() (in module fl)
tigetflag() (in module curses)
tigetnum() (in module curses)
tigetstr() (in module curses)
TILDE (in module token)
tilt() (in module turtle)
tiltangle() (in module turtle)
time (class in datetime)
    (module)
time() (datetime.datetime method)
    (in module time)
Time2Internaldate() (in module imaplib)
timedelta (class in datetime)
TimedRotatingFileHandler (class in logging.handlers)
traverseproc (C type)
Tree (class in Tix)
TreeBuilder (class in xml.etree.ElementTree)
Treeview (class in ttk)
triangular() (in module random)
triple-quoted string, [1]
True, [1], [2]
true
True (built-in variable)
truediv() (in module operator)
truncate() (in module math), [1]
truncate() (file method)
    (io.IOBase method)
truth
    value
truth() (in module operator)
try
    statement, [1], [2]
ttk
    (module)	tob() (in module imgfile)
-r N, --repeat=N
-s S, --setup=S
-t, --time
-v, --verbose
timeit() (in module timeit)
    (timeit.Timer method)
timeout
    (SocketServer.BaseServer attribute)
timeout() (curses.window method)
Timer (class in threading)
    (class in timeit)
times() (in module os)
timetuple() (datetime.date method)
    (datetime.datetime method)
timetz() (datetime.datetime method)
timezone (in module time)
title() (EasyDialogs.ProgressBar method)
    (in module turtle)
    (str method)
Tix
    (class in Tix)
    (module)
tix_addbitmapdir() (Tix.tixCommand method)
tty
    I/O control
tty (module)
tylabel() (in module os)
tuple
    built-in function, [1]
display
    empty, [1]
    object, [1], [2], [3], [4], [5]
singleton
tuple() (built-in function)
tuple2st() (in module parser)
tuple_params (2to3 fixer)
TupleType (in module types
    [1]
turnoff_sigfpe() (in module fpectl)
turnon_sigfpe() (in module fpectl)
Turtle (class in turtle)
turtle (module)
turtles() (in module turtle)
TurtleScreen (class in turtle)
turtlesize() (in module turtle)
Tutt, Bill
type, [1]
    Boolean
    built-in function, [1], [2], [3]
data
    hierarchy
immutable data
    object, [1], [2]
operations on dictionary
operations on list
type (built-in class)
Type (class in aetypes)
type (optparse.Option)
attribute
(socket.socket attribute)
(tarfile.TarInfo attribute)
type of an object
TYPE_CHECKER
(optparse.Option attribute)
typeahead() (in module
    curses)
typecode (array.array attribu-
    TYPED_ACTIONS
(optparse.Option attribute)
typed_subpart_iterator() (in
    module email.iterators)
TypeError
    exception
types
    built-in
    module
    mutable sequence
    operations on integer
    operations on mapping
    operations on numeric
    operations on sequence
[1]
types (2to3 fixer)
    (module)
TYPES (optparse.Option
    attribute)
types, internal
types_map (in module
    mimetypes)
    (mimetypes.MimeTypes
        attribute)
types_map_inv
    (mimetypes.MimeTypes
        attribute)
    TypeType (in module types)
TZ, [1], [2], [3], [4]
tzinfo (class in datetime)
  (datetime.datetime attribute)
  (datetime.time attribute)
tzname (in module time)
tzname() (datetime.datetime method)
  (datetime.time method)
  (datetime.tzinfo method)
tzset() (in module time)
U (in module re)
    u-LAW, [1], [2], [3]
ucd_3_2_0 (in module unicodedata)
udata (select.kevent attribute)
UDPServer (class in SocketServer)
UF_APPEND (in module stat)
UF_COMPRESSED (in module stat)
UF_HIDDEN (in module stat)
UF_IMMUTABLE (in module stat)
UF_NODUMP (in module stat)
UF_NOUNLINK (in module stat)
UF_OPAQUE (in module stat)
ufgettext()
    (gettext.GNUTranslations method)
    (gettext.NullTranslations method)
uid (tarfile.TarInfo attribute)
uid() (imaplib.IMAP4 method)
uidl() (poplib.POP3 method)
ulaw2lin() (in module audioop)
ULONG_MAX
uname() (in module os)
uname() (in module TarInfo attribute)
    (in module os)
    (in module platform)
unary
    arithmetic operation
    bitwise operation
UNARY_CONVERT (opcode)
UNARY_INVERT (opcode)
UNARY_NEGATIVE (opcode)
UNPACK_SEQUENCE (opcode)
unpack_string() (xdrlib.Unpacker method)
Unpacker (class in xdrlib)
unpackevent() (in module aetools)
uparseEntityDecl()
    (xml.sax.handler.DTDHandler method)
UnparsedEntityDeclHandler()
    (xml.parsers.expat.xmlparser method)
Unpickler (class in pickle)
UnpicklingError
unqdevice() (in module fl)
unquote() (in module email.utils)
    (in module rfc822)
    (in module urlib)
unquote_plus() (in module urlib)
unreachable object
unreadline() (distutils.text_file.TextFile method)
unrecognized escape sequence
unregister() (select.epoll method)
    (select.poll method)
unregister_archive_format() (in module shutil)
unregister_dialect() (in module csv)
unset()
    (test.test_support.EnvironmentVarGuard method)
unsetenv() (in module os)
unsubscribe() (imaplib.IMAP4 method)
UnsupportedOperation
untokenize() (in module tokenize)
untouchwin() (curses.window method)
unused_data (zlib.Decompress attribute)
UNARY_NOT (opcode)
UNARY_POSITIVE (opcode)
unbinding
  name
UnboundLocalError, [1]
UnboundMethodType (in module types)
unbuffered I/O
UNC paths
  and os.makedirs()
unconsumed_tail
(zlib.Decompress attribute)
unctrnl() (in module curses)
  (in module curses.ascii)
#undefine_macro()
(distutils.ccompiler.CCompiler method)
Underflow (class in decimal)
undo() (in module turtle)
undobufferentries() (in module turtle)
undoc_header (cmd.Cmd attribute)
unescape() (in module xml.sax.saxutils)
UnexpectedException
unexpectedSuccesses
(unittest.TestResult attribute)
unfreeze_form() (fl.form method)
ungetch() (in module curses)
  (in module msvcrt)
ungetmouse() (in module curses)
ungettext() (gettext.GNUTranslations method)
  (gettext.NullTranslations method)
ungetwch() (in module msvcrt)
unwrap() (ssl.SSLSocket method)
up() (in module turtle)
update() (collections.Counter method)
  (dict method)
  (hashlib.hash method)
  (hmac.HMAC method)
  (in module turtle)
(mailbox.Mailbox method)
(mailbox.Maildir method)
(md5.md5 method)
  (set method)
  (sha.sha method)
  (trace.CoverageResults method)
update_panels() (in module curses.panel)
update_visible()
(mailbox.BabylMessage method)
update_wrapper() (in module functools)
updatescrollbars()
(Framebuffer.ScrolledWindow method)
upper() (in module string)
  (str method)
upercase (in module string)
urandom() (in module os)
URL, [1], [2], [3], [4]
  parsing
  relative
url (xmlrpclib.ProtocolError attribute)
url2pathname() (in module urllib)
urlcleanup() (in module urllib)
urldefrag() (in module urllib)
urllib.parse() (in module urllib)
URLError
urljoin() (in module urllib)
urllib
  module
urllib (2to3 fixer)
unhexlify() (in module binascii)
unichr  
    built-in function
unichr() (built-in function)
Unicode, [1], [2]  
    database  
    object
unicode  
    built-in function, [1], [2], [3]  
    object
unicode (2to3 fixer)
UNICODE (in module re)
Unicode Consortium
unicode() (built-in function)
unicodedata (module)
UnicodeDecodeError
UnicodeEncodeError
UnicodeError
UnicodeTranslateError
UnicodeType (in module types)
UnicodeWarning
unidata_version (in module unicodedata)
unified_diff() (in module difflib)
uniform() (in module random)
UnimplementedFileMode
uninstall() (imputil.ImportManager method)
Union (class in ctypes)
union() (set method)
unittest (module)
unittest command line option  
    -b, --buffer
    -c, --catch
    -f, --failfast
unittest-discover command line option

urllib2 (module)
urlopen() (in module urllib)
    (in module urllib2)
URLopener (class in urllib)
urllist
    module
urllist (module)
urllist() (in module urlparse)
urllibretrieve() (in module urllib)
urllsafe_b64decode() (in module base64)
urllsafe_b64encode() (in module base64)
urllsplit() (in module urllib)
urllunparse() (in module urllib)
urllunsplit() (in module urllib)
urn (uuid.UUID attribute)
use_default_colors() (in module curses)
use_env() (in module curses)
use_rawinput (cmd.Cmd attribute)
UseForeignDTD()
(xml.parsers.expat.xmlparser method)
USER
user
    configuration file
    effective id
    id
    id, setting
user (module)
user() (poplib.Pop3 method)
user-defined
    function
    function call
    method
user-defined function
    object, [1], [2]
-p, --pattern pattern
-s, --start-directory directory
-t, --top-level-directory directory
-v, --verbose

universal newlines
What's new, [1], [2], [3]
bz2.BZ2File class
file.newlines attribute
io.IncrementalNewlineDecoder class
io.TextIOWrapper class
open() (in module io)
open() built-in function
str.splitlines method
subprocess module
zipfile.ZipFile.open method

UNIX
I/O control
file control
UnixDatagramServer (class in SocketServer)
unixfrom (rfc822.Message attribute)
UnixMailbox (class in mailbox)
UnixStreamServer (class in SocketServer)
Unknown (class in aetypes)
unknown_charref()
(sgmllib.SGMLParser method)
unknown_decl()
(HTMLParser.HTMLParser method)
unknown_endtag()
(sgmllib.SGMLParser method)
unknown_entityref()

user-defined method

object

USER_BASE
(in module site)
user_call() (bdb.Bdb method)
user_exception() (bdb.Bdb method)
user_line() (bdb.Bdb method)
user_return() (bdb.Bdb method)
USER_SITE (in module site)

usercustomize
module

UserDict (class in UserDict)
(module)

UserList (class in UserList)
(module)

USERNAME
USERPROFILE, [1]
userptr() (curses.panel.Panel method)

UserString (class in UserString)
(module)

UserWarning

USTAR_FORMAT (in module tarfile)
UTC
utcfromtimestamp() (datetime.datetime method)
utcnow() (datetime.datetime class method)

utcoffset() (datetime.datetime method)
(utcoffset().method)

utctimetuple() (datetime.datetime method)

utime() (in module os)
uu
(module)
uu (module)

UUID (class in uuid)
uuid (module)
uuid1
uuid1() (in module uuid)
uuid3
uuid3() (in module uuid)
uuid4
uuid4() (in module uuid)
uuid5
uuid5() (in module uuid)
UuidCreate() (in module msilib)

unknown_open() (urllib2.BaseHandler method)
unknown_starttag() (sgmllib.SGMLParser method)
UnknownHandler (class in urllib2)
UnknownProtocol
UnknownTransferEncoding
unlink() (in module os)
unlock() (mailbox.Babyl method)
unlock() (mailbox.MH method)
unlock() (mailbox.MMDF method)
unlock() (mailbox.Mailbox method)
unlock() (mailbox.Maildir method)
unlock() (mailbox.mbox method)
unlock() (mutex.mutex method)
unmimify() (in module mimify)
unpack() (in module aepack)
unpack() (in module struct)
unpack_array() (xdrlib.Unpacker method)
unpack_bytes() (xdrlib.Unpacker method)
unpack_double() (xdrlib.Unpacker method)
unpack_farray() (xdrlib.Unpacker method)
unpack_float() (xdrlib.Unpacker method)
unpack_fopaque() (xdrlib.Unpacker method)
unpack_from() (in module struct)
(struct.Struct method)
unpack_fstring() (xdrlib.Unpacker method)
unpack_list() (xdrlib.Unpacker method)
unpack_opaque()
(xdrlib.Unpacker method)
validator() (in module wsgiref.validate)

value
  default parameter
  truth
value (Cookie.Morsel attribute)
  (cookielib.Cookie attribute)
  (ctypes._SimpleCData attribute)
  (xml.dom.Attr attribute)
value of an object
Value() (in module multiprocessing)
  (in module multiprocessing.sharedctypes)
  (multiprocessing.managers.SyncManager method)
value_decode() (Cookie.BaseCookie method)
value_encode() (Cookie.BaseCookie method)
ValueError
  exception
valuerefs() (weakref.WeakValueDictionary method)
values
  Boolean
  writing, [1]
values() (dict method)
  (email.message.Message method)
  (mailbox.Mailbox method)
ValuesView (class in collections)
variable
  free, [1]
variant (uuid.UUID attribute)
varray() (in module gl)
vars() (built-in function)

VERIFY_DEFAULT (in module ssl)
verify_flags (ssl.SSLContext attribute)
verify_mode (ssl.SSLContext attribute)
verify_request() (SocketServer.BaseServer method)
VERIFY_X509 STRICT
VERIFY_X509 TRUSTED_FIRST (ssl)
version (cookielib.Cookie method)
  (httplib.HTTPResponse method)
  (in module curses)
  (in module marshal)
  (in module sqlite3)
  (in module sys), [1], [2]
  (urllib.URLopener attribute)
  (uuid.UUID attribute)
version() (in module sys)
  (in module platform)
  (ssl.SSLSocket method)
version_info (in module sqlite3)
  (in module sys)
version_string() (BaseHTTPServer.BaseHTTPRequestHandler method)
vformat() (string.Formatter method)
videoreader (module)
viewitems() (dict method)
viewkeys() (dict method)
viewvalues() (dict method)
virtual environment
virtual machine
visit() (ast.NodeVisitor method)
visitproc (C type)
vline() (curses.window method)
VBAR (in module token)

vbar (ScrolledText.ScrolledText attribute)

VBAREQUAL (in module token)

Vec2D (class in turtle)

VERBOSE (in module re)

verbose (in module tabnanny)

    (in module test.test_support)

verify() (smtplib.SMTP method)

VERIFY_CRL_CHECK_CHAIN (in module ssl)

VERIFY_CRL_CHECK_LEAF (in module ssl)

VMSError

vnarray() (in module gl)

voidcmd() (ftplib.FTP method)

volume (zipfile.ZipInfo attribute)

vonmisesvariate() (in module math)
W (module)
W_OK (in module os)
wait() (in module os)
  (multiprocessing.pool.AsyncResult method)
  (popen2.Popen3 method)
  (subprocess.Popen method)
  (threading.Condition method)
  (threading.Event method)
wait3() (in module os)
wait4() (in module os)
waitpid() (in module os)
walk() (email.message.Message method)
  (in module ast)
  (in module compiler)
  (in module compiler.visitor)
  (in module os)
  (in module os.path)
walk_packages() (in module pkgutil)
want (doctest.Example attribute)
warn() (distutils.ccompiler.CCompiler method)
  (distutils.text_file.TextFile method)
  (in module warnings)
warn_explicit() (in module warnings)
Warning
warning() (in module logging)
  (logging.Logger method)
  (xml.sax.handler.ErrorHandler method)
warnings
  (module)
WarningsRecorder (class in test.test_support)
warnoptions (in module sys)
warnpy3k() (in module warnings)

WinSock

winsound (module)

winver (in module sys)

with
  statement, [1]
WITH_CLEANUP (opcodes)
WMAvailable() (in module MacOS)
WNOHANG (in module os)
wordchars (shlex.shlex attribute)
World Wide Web, [1], [2]
wrap() (in module textwrap)
  (textwrap.TextWrapper method)
wrap_socket() (in module ssl)
  (ssl.SSLContext method)
wrap_text() (in module distutils.fancy_getopt)
wrapper() (in module urllib.parse)
writable() (asyncore.dispatcher method)
  (io.IOBase method)
write() (array.array method)
  (ConfigParser.RawConfigParser method)
  (bz2.BZ2File method)
  (code.InteractiveInterpreter method)
  (codecs.StreamWriter method)
  (email.generator.Generator method)
  (file method)
writeable() (asyncore.dispatcher method)
  (io.BufferedIOBase method)

with statement, [1]
wasSuccessful() (unittest.TestResult method)
WatchedFileHandler (class in
logging.handlers)
wave (module)
WCONTINUED (in module os)
WCOREDUMP() (in module os)
WeakKeyDictionary (class in weakref)
weakref (module)
WeakSet (class in weakref)
WeakValueDictionary (class in weakref)
webbrowser (module)
weekday() (datetime.date method)
   (datetime.datetime method)
   (in module calendar)
weekheader() (in module calendar)
weibullvariate() (in module random)
WEXITSTATUS() (in module os)
wfile
(BaseHTTPServer.BaseHTTPRequestHandler
attribute)
what() (in module imghdr)
   (in module sndhdr)
whathdr() (in module sndhdr)
whichdb (module)
whichdb() (in module whichdb)
while
   statement, [1], [2], [3]
whitespace
   (in module string)
   (shlex.shlex attribute)
whitespace_split (shlex.shlex attribute)
whseed() (in module random)
WichmannHill (class in random)
Widget (class in ttk)
width (textwrap.TextWrapper attribute)
width() (in module turtle)
WIFCONTINUED() (in module os)
WIFEXITED() (in module os)
WIFSIGNALED() (in module os)
WIFSTOPPED() (in module os)
Wimp$ScrapDir
win32_ver() (in module platform)
WinDLL (class in ctypes)
window manager (widgets)
window() (curses.panel.Panel method)
Window() (in module FrameWork)
window_height() (in module turtle), [1]
window_width() (in module turtle), [1]
windowbounds() (in module FrameWork)
Windows ini file
WindowsError
WinError() (in module ctypes)
WINFUNCTYPE() (in module ctypes)
writepy() (zipfile.PyZipFile method)
writer (formatter.formatter attribute)
writer() (module csv)
writerow() (csv.csvwriter method)
writerows() (csv.csvwriter method)
writestr() (zipfile.ZipFile method)
writexml() (xml.dom.minidom method)
writing values, [1]
WrongDocumentErr
ws_comma (2to3 fixer)
wsgi_file_wrapper
(wsgiref.handlers.Base
wsgi_multiprocess
(wsgiref.handlers.Base
wsgi_multithread
(wsgiref.handlers.Base
wsgi_run_once
(wsgiref.handlers.Base
wsgiref (module)
wsgiref.handlers (module)
wsgiref.headers (module)
wsgiref.simple_server (module)
wsgiref.util (module)
wsgiref.validate (module)
WSGIRequestHandler (class in wsgiref.simple_server)
WSGIServer (class in wsgiref.simple_server)
wShowWindow (subprocess.STARTUPINFO)
WSTOPSIG() (in module os)
wstring_at() (in module ctypes)
WTERMSIG() (in module os)
WUNTRACED (in module os)
WWW, [1], [2], [3]
server, [1]
X (in module re)  XML_NAMESPACE (in module xml.dom)
X509 certificate xmlcharrefreplace_errors() (in module codecs)
X_OK (in module os) XmlDeclHandler()
xatom() (imaplib.IMAP4 XDR, [1]) (xml.parsers.expat.xmlparser method)
xcor() (in module turtle) XMLGenerator (class in xml.sax.saxutils)
XDR, [1] XMLFilterBase (class in xml.sax.saxutils)
xdrlib (module) xgtitle() (nntplib.NNTP method) XMLID() (in module xml.etree.ElementTree)
xgtitle() (nntplib.NNTP method) XMLNS_NAMESPACE (in module xml.dom)
xhdr() (nntplib.NNTP method) XMLParser (class in xml.etree.ElementTree)
XHTML XMLParserType (in module xml.parsers.expat)
XHTML_NAMESPACE (in module xml.dom) XMLReader (class in xml.sax.xmlreader)
xml (module) xmlrpclib (module)
XML() (in module XMLGenerator (class in xml.parsers.expat.xmlparser)
xml.etree.ElementTree) XMLFilterBase (class in xml.sax.saxutils)
xml.dom (module) XMLID() (in module xml.etree.ElementTree)
xml.dom.minidom (module) XMLNS_NAMESPACE (in module xml.dom)
xml.dom.pulldom (module) XMLParser (class in xml.etree.ElementTree)
xmletree.ElementTree XMLParserType (in module xml.parsers.expat)
xrange(2to3 fixer) XMLReader (class in xml.sax.xmlreader)
xrange() (built-in function)
xml.etree.ElementTree XRangeType (in module types)
xrangeType (in module types)
xreadlines (2to3 fixer) xreadlines() (bz2.BZ2File method)
xreadlines() (bz2.BZ2File method)
xrange
 xor
    bitwise
 xor() (in module operator)
 xover() (nntplib.NNTP method)
xpath() (nntplib.NNTP method)
xrange
    built-in function
    object, [1]
xrange (2to3 fixer) xreadlines (2to3 fixer)
(file method)
xview() (ttk.Treeview method)
Y

Y2K
ycor() (in module turtle)
year (datetime.date attribute)
    (datetime.datetime attribute)
Year 2000
Year 2038
yeardatescalendar()
    (calendar.Calendar method)
yeardays2calendar()
    (calendar.Calendar method)
yeardayscalendar()
    (calendar.Calendar method)
YESEXPR (in module locale)
yield
    expression
    keyword
    statement
YIELD_VALUE (opcode)
yiq_to_rgb() (in module colorsys)
yview() (ttk.Treeview method)
Zen of Python
ZeroDivisionError
  exception
zfill() (in module string)
    (str method)
zip (2to3 fixer)
zip() (built-in function)
    (in module future_builtins)
ZIP_DEFLATED (in module zipfile)
ZIP_STORED (in module zipfile)
ZipFile (class in zipfile)
zipfile (module)
  zipfile command line option
    -c <zipfile> <source1> ...
    <sourceN>
    -e <zipfile> <output_dir>
    -I <zipfile>
    -t <zipfile>
zipimport (module)
zipimporter (class in zipimport)
ZipImportError
ZipInfo (class in zipfile)
zlib (module)
9. Command Reference
9.1. Installing modules: the **install** command family

The install command ensures that the build commands have been run and then runs the subcommands `install_lib`, `install_data` and `install_scripts`.

### 9.1.1. `install_data`

This command installs all data files provided with the distribution.

### 9.1.2. `install_scripts`

This command installs all (Python) scripts in the distribution.
3. Writing the Setup Configuration File

Often, it's not possible to write down everything needed to build a distribution \textit{a priori}: you may need to get some information from the user, or from the user's system, in order to proceed. As long as that information is fairly simple—a list of directories to search for C header files or libraries, for example—then providing a configuration file, \texttt{setup.cfg}, for users to edit is a cheap and easy way to solicit it. Configuration files also let you provide default values for any command option, which the installer can then override either on the command-line or by editing the config file.

The setup configuration file is a useful middle-ground between the setup script—which, ideally, would be opaque to installers—\cite{[1]}—and the command-line to the setup script, which is outside of your control and entirely up to the installer. In fact, \texttt{setup.cfg} (and any other Distutils configuration files present on the target system) are processed after the contents of the setup script, but before the command-line. This has several useful consequences:

- installers can override some of what you put in \texttt{setup.py} by editing \texttt{setup.cfg}
- you can provide non-standard defaults for options that are not easily set in \texttt{setup.py}
- installers can override anything in \texttt{setup.cfg} using the command-line options to \texttt{setup.py}

The basic syntax of the configuration file is simple:

```
[command]
option=value
...
```
where command is one of the Distutils commands (e.g. `build_py`, `install`), and option is one of the options that command supports. Any number of options can be supplied for each command, and any number of command sections can be included in the file. Blank lines are ignored, as are comments, which run from a `#` character until the end of the line. Long option values can be split across multiple lines simply by indenting the continuation lines.

You can find out the list of options supported by a particular command with the universal `--help` option, e.g.

```bash
> python setup.py --help build_ext
[...]
Options for 'build_ext' command:
   --build-lib (-b) directory for compiled extension modules
   --build-temp (-t) directory for temporary files (build by-
                     ignoring build-lib and put compiled extens
                     source directory alongside your pure Pyt
   --inplace (-i) ignore build-lib and put compiled extens
   --include-dirs (-I) list of directories to search for header
   --define (-D) C preprocessor macros to define
   --undef (-U) C preprocessor macros to undefine
   --swig-opts list of SWIG command line options
[...]
```

Note that an option spelled `--foo-bar` on the command-line is spelled `foo_bar` in configuration files.

For example, say you want your extensions to be built “in-place”—that is, you have an extension `pkg.ext`, and you want the compiled extension file (`ext.so` on Unix, say) to be put in the same source directory as your pure Python modules `pkg.mod1` and `pkg.mod2`. You can always use the `--inplace` option on the command-line to ensure this:

```bash
python setup.py build_ext --inplace
```

But this requires that you always specify the `build_ext` command
explicitly, and remember to provide --inplace. An easier way is to "set and forget" this option, by encoding it in setup.cfg, the configuration file for this distribution:

```ini
[build_ext]
inplace=1
```

This will affect all builds of this module distribution, whether or not you explicitly specify build_ext. If you include setup.cfg in your source distribution, it will also affect end-user builds—which is probably a bad idea for this option, since always building extensions in-place would break installation of the module distribution. In certain peculiar cases, though, modules are built right in their installation directory, so this is conceivably a useful ability. (Distributing extensions that expect to be built in their installation directory is almost always a bad idea, though.)

Another example: certain commands take a lot of options that don’t change from run to run; for example, bdist_rpm needs to know everything required to generate a “spec” file for creating an RPM distribution. Some of this information comes from the setup script, and some is automatically generated by the Distutils (such as the list of files installed). But some of it has to be supplied as options to bdist_rpm, which would be very tedious to do on the command-line for every run. Hence, here is a snippet from the Distutils’ own setup.cfg:

```ini
[bdist_rpm]
release = 1
packager = Greg Ward <gward@python.net>
doc_files = CHANGEs.txt
          README.txt
          USAGE.txt
doc/
    examples/
```
Note that the `doc_files` option is simply a whitespace-separated string split across multiple lines for readability.

**See also:**

**Syntax of config files** in “Installing Python Modules”

More information on the configuration files is available in the manual for system administrators.

**Footnotes**

[1] This ideal probably won't be achieved until auto-configuration is fully supported by the Distutils.
Distributing Python Modules (Legacy version) »
2. Writing the Setup Script

The setup script is the centre of all activity in building, distributing, and installing modules using the Distutils. The main purpose of the setup script is to describe your module distribution to the Distutils, so that the various commands that operate on your modules do the right thing. As we saw in section A Simple Example above, the setup script consists mainly of a call to setup(), and most information supplied to the Distutils by the module developer is supplied as keyword arguments to setup().

Here's a slightly more involved example, which we'll follow for the next couple of sections: the Distutils' own setup script. (Keep in mind that although the Distutils are included with Python 1.6 and later, they also have an independent existence so that Python 1.5.2 users can use them to install other module distributions. The Distutils' own setup script, shown here, is used to install the package into Python 1.5.2.)

```python
#!/usr/bin/env python

from distutils.core import setup

setup(name='Distutils',
      version='1.0',
      description='Python Distribution Utilities',
      author='Greg Ward',
      author_email='gward@python.net',
      url='https://www.python.org/sigs/distutils-sig/',
      packages=['distutils', 'distutils.command'],
)
```

There are only two differences between this and the trivial one-file distribution presented in section A Simple Example: more metadata, and the specification of pure Python modules by package, rather than by module. This is important since the Distutils consist of a
couple of dozen modules split into (so far) two packages; an explicit list of every module would be tedious to generate and difficult to maintain. For more information on the additional meta-data, see section Additional meta-data.

Note that any pathnames (files or directories) supplied in the setup script should be written using the Unix convention, i.e. slash-separated. The Distutils will take care of converting this platform-neutral representation into whatever is appropriate on your current platform before actually using the pathname. This makes your setup script portable across operating systems, which of course is one of the major goals of the Distutils. In this spirit, all pathnames in this document are slash-separated.

This, of course, only applies to pathnames given to Distutils functions. If you, for example, use standard Python functions such as `glob.glob()` or `os.listdir()` to specify files, you should be careful to write portable code instead of hardcoding path separators:

```python
import os

glob.glob(os.path.join('mydir', 'subdir', '*.html'))
os.listdir(os.path.join('mydir', 'subdir'))
```
2.1. Listing whole packages

The `packages` option tells the Distutils to process (build, distribute, install, etc.) all pure Python modules found in each package mentioned in the `packages` list. In order to do this, of course, there has to be a correspondence between package names and directories in the filesystem. The default correspondence is the most obvious one, i.e. package `distutils` is found in the directory `distutils` relative to the distribution root. Thus, when you say `packages = ['foo']` in your setup script, you are promising that the Distutils will find a file `foo/__init__.py` (which might be spelled differently on your system, but you get the idea) relative to the directory where your setup script lives. If you break this promise, the Distutils will issue a warning but still process the broken package anyway.

If you use a different convention to lay out your source directory, that's no problem: you just have to supply the `package_dir` option to tell the Distutils about your convention. For example, say you keep all Python source under `lib`, so that modules in the “root package” (i.e., not in any package at all) are in `lib`, modules in the `foo` package are in `lib/foo`, and so forth. Then you would put

```python
package_dir = {'': 'lib'}
```

in your setup script. The keys to this dictionary are package names, and an empty package name stands for the root package. The values are directory names relative to your distribution root. In this case, when you say `packages = ['foo']`, you are promising that the file `lib/foo/__init__.py` exists.

Another possible convention is to put the `foo` package right in `lib`,
the foo.bar package in lib/bar, etc. This would be written in the setup script as

```python
package_dir = {'foo': 'lib'}
```

A package: dir entry in the package_dir dictionary implicitly applies to all packages below package, so the foo.bar case is automatically handled here. In this example, having `packages = ['foo', 'foo.bar']` tells the Distutils to look for `lib/__init__.py` and `lib/bar/__init__.py`. (Keep in mind that although package_dir applies recursively, you must explicitly list all packages in packages: the Distutils will not recursively scan your source tree looking for any directory with an __init__.py file.)
2.2. Listing individual modules

For a small module distribution, you might prefer to list all modules rather than listing packages—especially the case of a single module that goes in the “root package” (i.e., no package at all). This simplest case was shown in section A Simple Example; here is a slightly more involved example:

```python
py_modules = ['mod1', 'pkg.mod2']
```

This describes two modules, one of them in the “root” package, the other in the `pkg` package. Again, the default package/directory layout implies that these two modules can be found in `mod1.py` and `pkg/mod2.py`, and that `pkg/__init__.py` exists as well. And again, you can override the package/directory correspondence using the `package_dir` option.
2.3. Describing extension modules

Just as writing Python extension modules is a bit more complicated than writing pure Python modules, describing them to the Distutils is a bit more complicated. Unlike pure modules, it's not enough just to list modules or packages and expect the Distutils to go out and find the right files; you have to specify the extension name, source file(s), and any compile/link requirements (include directories, libraries to link with, etc.).

All of this is done through another keyword argument to `setup()`, the `ext_modules` option. `ext_modules` is just a list of `Extension` instances, each of which describes a single extension module. Suppose your distribution includes a single extension, called `foo` and implemented by `foo.c`. If no additional instructions to the compiler/linker are needed, describing this extension is quite simple:

```python
Extension('foo', ['foo.c'])
```

The `Extension` class can be imported from `distutils.core` along with `setup()`. Thus, the setup script for a module distribution that contains only this one extension and nothing else might be:

```python
from distutils.core import setup, Extension
setup(name='foo',
      version='1.0',
      ext_modules=[Extension('foo', ['foo.c'])],
)
```

The `Extension` class (actually, the underlying extension-building machinery implemented by the `build_ext` command) supports a great deal of flexibility in describing Python extensions, which is explained in the following sections.
2.3.1. Extension names and packages

The first argument to the `Extension` constructor is always the name of the extension, including any package names. For example,

```python
Extension('foo', ['src/foo1.c', 'src/foo2.c'])
```

describes an extension that lives in the root package, while

```python
Extension('pkg.foo', ['src/foo1.c', 'src/foo2.c'])
```

describes the same extension in the `pkg` package. The source files and resulting object code are identical in both cases; the only difference is where in the filesystem (and therefore where in Python’s namespace hierarchy) the resulting extension lives.

If you have a number of extensions all in the same package (or all under the same base package), use the `ext_package` keyword argument to `setup()`. For example,

```python
setup(...,
    ext_package='pkg',
    ext_modules=[Extension('foo', ['foo.c']),
                 Extension('subpkg.bar', ['bar.c'])],
)
```

will compile `foo.c` to the extension `pkg.foo`, and `bar.c` to `pkg.subpkg.bar`.

2.3.2. Extension source files

The second argument to the `Extension` constructor is a list of source files. Since the Distutils currently only support C, C++, and Objective-C extensions, these are normally C/C++/Objective-C source files. (Be sure to use appropriate extensions to distinguish
C++ source files: .cc and .cpp seem to be recognized by both Unix and Windows compilers.)

However, you can also include SWIG interface (.i) files in the list; the build_ext command knows how to deal with SWIG extensions: it will run SWIG on the interface file and compile the resulting C/C++ file into your extension.

This warning notwithstanding, options to SWIG can be currently passed like this:

```python
setup(...,
    ext_modules=[Extension('_foo', ['foo.i'],
                             swig_opts=['-modern', '-I../include'],
                             py_modules=['foo'],
    )
```

Or on the commandline like this:

```
> python setup.py build_ext --swig-opts="-modern -I../include"
```

On some platforms, you can include non-source files that are processed by the compiler and included in your extension. Currently, this just means Windows message text (.mc) files and resource definition (.rc) files for Visual C++. These will be compiled to binary resource (.res) files and linked into the executable.

### 2.3.3. Preprocessor options

Three optional arguments to `Extension` will help if you need to specify include directories to search or preprocessor macros to define/undefine: `include_dirs`, `define_macros`, and `undef_macros`.

For example, if your extension requires header files in the `include` directory under your distribution root, use the `include_dirs` option:
You can specify absolute directories there; if you know that your extension will only be built on Unix systems with X11R6 installed to \texttt{/usr}, you can get away with

Extension(\texttt{\'foo\', ['foo.c'], include_dirs=[\texttt{\'/usr/include/X11\']})

You should avoid this sort of non-portable usage if you plan to distribute your code: it's probably better to write C code like

\texttt{\#include <X11/Xlib.h>}

If you need to include header files from some other Python extension, you can take advantage of the fact that header files are installed in a consistent way by the Distutils \texttt{install\_headers} command. For example, the Numerical Python header files are installed (on a standard Unix installation) to \texttt{/usr/local/include/python1.5/Numerical}. (The exact location will differ according to your platform and Python installation.) Since the Python include directory—\texttt{/usr/local/include/python1.5} in this case—is always included in the search path when building Python extensions, the best approach is to write C code like

\texttt{\#include <Numerical/arrayobject.h>}

If you must put the \texttt{Numerical} include directory right into your header search path, though, you can find that directory using the Distutils \texttt{distutils.sysconfig} module:

\texttt{from distutils.sysconfig import get\textunderscore python\textunderscore inc
inicdir = os.path.join(get\textunderscore python\textunderscore inc(plat\textunderscore specific=1), \texttt{\'Numerical\'})
setup(\ldots,
     Extension(\ldots, include\textunderscore dirs=[inicdir]),
    )}
Even though this is quite portable—it will work on any Python installation, regardless of platform—it’s probably easier to just write your C code in the sensible way.

You can define and undefine pre-processor macros with the `define_macros` and `undef_macros` options. `define Macros` takes a list of `(name, value)` tuples, where `name` is the name of the macro to define (a string) and `value` is its value: either a string or `None`. (Defining a macro `FOO` to `None` is the equivalent of a bare `#define FOO` in your C source: with most compilers, this sets `FOO` to the string `1`.)

`undef Macros` is just a list of macros to redefine.

For example:

```python
Extension(...,
    define_macros=[('NDEBUG', '1'),
                  ('HAVE_STRFTIME', None)],
    undef_macros=['HAVE_FOO', 'HAVE_BAR'])
```

is the equivalent of having this at the top of every C source file:

```c
#define NDEBUG 1
#define HAVE_STRFTIME
#undef HAVE_FOO
#undef HAVE_BAR
```

### 2.3.4. Library options

You can also specify the libraries to link against when building your extension, and the directories to search for those libraries. The `libraries` option is a list of libraries to link against, `library_dirs` is a list of directories to search for libraries at link-time, and `runtime_library_dirs` is a list of directories to search for shared (dynamically loaded) libraries at run-time.

For example, if you need to link against libraries known to be in the
standard library search path on target systems

```python
Extension(...,
    libraries=['gdbm', 'readline'])
```

If you need to link with libraries in a non-standard location, you'll have to include the location in `library_dirs`:

```python
Extension(...,
    library_dirs=['/usr/X11R6/lib'],
    libraries=['X11', 'Xt'])
```

(Again, this sort of non-portable construct should be avoided if you intend to distribute your code.)

### 2.3.5. Other options

There are still some other options which can be used to handle special cases.

The `extra_objects` option is a list of object files to be passed to the linker. These files must not have extensions, as the default extension for the compiler is used.

`extra_compile_args` and `extra_link_args` can be used to specify additional command line options for the respective compiler and linker command lines.

`export_symbols` is only useful on Windows. It can contain a list of symbols (functions or variables) to be exported. This option is not needed when building compiled extensions: Distutils will automatically add `initmodule` to the list of exported symbols.

The `depends` option is a list of files that the extension depends on (for example header files). The build command will call the compiler on the sources to rebuild extension if any on this files has been modified.
since the previous build.
2.4. Relationships between Distributions and Packages

A distribution may relate to packages in three specific ways:

1. It can require packages or modules.
2. It can provide packages or modules.
3. It can obsolete packages or modules.

These relationships can be specified using keyword arguments to the `distutils.core.setup()` function.

Dependencies on other Python modules and packages can be specified by supplying the `requires` keyword argument to `setup()`. The value must be a list of strings. Each string specifies a package that is required, and optionally what versions are sufficient.

To specify that any version of a module or package is required, the string should consist entirely of the module or package name. Examples include `mymodule` and `xml.parsers.expat`.

If specific versions are required, a sequence of qualifiers can be supplied in parentheses. Each qualifier may consist of a comparison operator and a version number. The accepted comparison operators are:

```
<   >   ==
<=  >=  !=
```

These can be combined by using multiple qualifiers separated by commas (and optional whitespace). In this case, all of the qualifiers must be matched; a logical AND is used to combine the evaluations.

Let’s look at a bunch of examples:
<table>
<thead>
<tr>
<th>Requires Expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>==1.0</td>
<td>Only version 1.0 is compatible</td>
</tr>
<tr>
<td>&gt;1.0, !=1.5.1, &lt;2.0</td>
<td>Any version after 1.0 and before 2.0 is compatible, except 1.5.1</td>
</tr>
</tbody>
</table>

Now that we can specify dependencies, we also need to be able to specify what we provide that other distributions can require. This is done using the `provides` keyword argument to `setup()`. The value for this keyword is a list of strings, each of which names a Python module or package, and optionally identifies the version. If the version is not specified, it is assumed to match that of the distribution.

Some examples:

<table>
<thead>
<tr>
<th>Provides Expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>mypkg</td>
<td>Provide mypkg, using the distribution version</td>
</tr>
<tr>
<td>mypkg (1.1)</td>
<td>Provide mypkg version 1.1, regardless of the distribution version</td>
</tr>
</tbody>
</table>

A package can declare that it obsoletes other packages using the `obsoletes` keyword argument. The value for this is similar to that of the `requires` keyword: a list of strings giving module or package specifiers. Each specifier consists of a module or package name optionally followed by one or more version qualifiers. Version qualifiers are given in parentheses after the module or package name.

The versions identified by the qualifiers are those that are obsoleted by the distribution being described. If no qualifiers are given, all versions of the named module or package are understood to be obsoleted.
2.5. Installing Scripts

So far we have been dealing with pure and non-pure Python modules, which are usually not run by themselves but imported by scripts.

Scripts are files containing Python source code, intended to be started from the command line. Scripts don’t require Distutils to do anything very complicated. The only clever feature is that if the first line of the script starts with `#!` and contains the word “python”, the Distutils will adjust the first line to refer to the current interpreter location. By default, it is replaced with the current interpreter location. The **--executable** (or `-e`) option will allow the interpreter path to be explicitly overridden.

The `scripts` option simply is a list of files to be handled in this way. From the PyXML setup script:

```python
setup(...,
      scripts=['scripts/xmlproc_parse', 'scripts/xmlproc_val'])
```

*Changed in version 2.7:* All the scripts will also be added to the MANIFEST file if no template is provided. See Specifying the files to distribute.
2.6. Installing Package Data

Often, additional files need to be installed into a package. These files are often data that’s closely related to the package’s implementation, or text files containing documentation that might be of interest to programmers using the package. These files are called *package data*.

Package data can be added to packages using the `package_data` keyword argument to the `setup()` function. The value must be a mapping from package name to a list of relative path names that should be copied into the package. The paths are interpreted as relative to the directory containing the package (information from the `package_dir` mapping is used if appropriate); that is, the files are expected to be part of the package in the source directories. They may contain glob patterns as well.

The path names may contain directory portions; any necessary directories will be created in the installation.

For example, if a package should contain a subdirectory with several data files, the files can be arranged like this in the source tree:

```plaintext
setup.py
src/
  mypkg/
    __init__.py
    module.py
  data/
    tables.dat
    spoons.dat
    forks.dat
```

The corresponding call to `setup()` might be:

```python
setup(...,
```
New in version 2.4.

Changed in version 2.7: All the files that match `package_data` will be added to the `MANIFEST` file if no template is provided. See Specifying the files to distribute.
The `data_files` option can be used to specify additional files needed by the module distribution: configuration files, message catalogs, data files, anything which doesn’t fit in the previous categories.

`data_files` specifies a sequence of `(directory, files)` pairs in the following way:

```python
setup(...,
    data_files=[('bitmaps', ['bm/b1.gif', 'bm/b2.gif'])),
           ('config', ['cfg/data.cfg'])),
           ('etc/init.d', ['init-script']))
)
```

Note that you can specify the directory names where the data files will be installed, but you cannot rename the data files themselves.

Each `(directory, files)` pair in the sequence specifies the installation directory and the files to install there. If `directory` is a relative path, it is interpreted relative to the installation prefix (Python’s `sys.prefix` for pure-Python packages, `sys.exec_prefix` for packages that contain extension modules). Each file name in `files` is interpreted relative to the `setup.py` script at the top of the package source distribution. No directory information from `files` is used to determine the final location of the installed file; only the name of the file is used.

You can specify the `data_files` options as a simple sequence of files without specifying a target directory, but this is not recommended, and the `install` command will print a warning in this case. To install data files directly in the target directory, an empty string should be given as the directory.

*Changed in version 2.7:* All the files that match `data_files` will be
added to the MANIFEST file if no template is provided. See Specifying the files to distribute.
2.8. Additional meta-data

The setup script may include additional meta-data beyond the name and version. This information includes:

<table>
<thead>
<tr>
<th>Meta-Data</th>
<th>Description</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name of the package</td>
<td>short string</td>
<td>(1)</td>
</tr>
<tr>
<td>version</td>
<td>version of this release</td>
<td>short string</td>
<td>(1)(2)</td>
</tr>
<tr>
<td>author</td>
<td>package author’s name</td>
<td>short string</td>
<td>(3)</td>
</tr>
<tr>
<td>author_email</td>
<td>email address of the package author</td>
<td>email address</td>
<td>(3)</td>
</tr>
<tr>
<td>maintainer</td>
<td>package maintainer’s name</td>
<td>short string</td>
<td>(3)</td>
</tr>
<tr>
<td>maintainer_email</td>
<td>email address of the package maintainer</td>
<td>email address</td>
<td>(3)</td>
</tr>
<tr>
<td>url</td>
<td>home page for the package</td>
<td>URL</td>
<td>(1)</td>
</tr>
<tr>
<td>description</td>
<td>short, summary description of the package</td>
<td>short string</td>
<td></td>
</tr>
<tr>
<td>long_description</td>
<td>longer description of the package</td>
<td>long string</td>
<td>(5)</td>
</tr>
<tr>
<td>download_url</td>
<td>location where the package may be downloaded</td>
<td>URL</td>
<td>(4)</td>
</tr>
<tr>
<td>classifiers</td>
<td>a list of classifiers</td>
<td>list of strings</td>
<td>(4)</td>
</tr>
<tr>
<td>platforms</td>
<td>a list of platforms</td>
<td>list of strings</td>
<td></td>
</tr>
<tr>
<td>license</td>
<td>license for the package</td>
<td>short string</td>
<td>(6)</td>
</tr>
</tbody>
</table>

Notes:

1. These fields are required.
2. It is recommended that versions take the form
Encoding the version information is an art in itself. Python packages generally adhere to the version format *major.minor[.patch][sub]*. The major number is 0 for initial, experimental releases of software. It is incremented for releases that represent major milestones in a package. The minor number is incremented when important new features are added to the package. The patch number increments when bug-fix releases are made. Additional trailing version information is sometimes used to indicate sub-releases. These are “a1,a2,...,aN” (for alpha releases, where functionality and API may
change), “b1,b2,...,bN” (for beta releases, which only fix bugs) and “pr1,pr2,...,prN” (for final pre-release release testing). Some examples:

0.1.0
   the first, experimental release of a package

1.0.1a2
   the second alpha release of the first patch version of 1.0

classifiers are specified in a Python list:

```
setup(...,
   classifiers=[
      'Development Status :: 4 - Beta',
      'Environment :: Console',
      'Environment :: Web Environment',
      'Intended Audience :: End Users/Desktop',
      'Intended Audience :: Developers',
      'Intended Audience :: System Administrators',
      'License :: OSI Approved :: Python Software Foundation',
      'Operating System :: MacOS :: MacOS X',
      'Operating System :: Microsoft :: Windows',
      'Operating System :: POSIX',
      'Programming Language :: Python',
      'Topic :: Communications :: Email',
      'Topic :: Office/Business',
      'Topic :: Software Development :: Bug Tracking',
   ],
)
```

If you wish to include classifiers in your `setup.py` file and also wish to remain backwards-compatible with Python releases prior to 2.2.3, then you can include the following code fragment in your `setup.py` before the `setup()` call.

```
# patch distutils if it can't cope with the "classifiers" or
# "download_url" keywords
from sys import version
if version < '2.2.3':

```
from distutils.dist import DistributionMetadata
DistributionMetadata.classifiers = None
DistributionMetadata.download_url = None
2.9. Debugging the setup script

Sometimes things go wrong, and the setup script doesn't do what the developer wants.

Distutils catches any exceptions when running the setup script, and print a simple error message before the script is terminated. The motivation for this behaviour is to not confuse administrators who don't know much about Python and are trying to install a package. If they get a big long traceback from deep inside the guts of Distutils, they may think the package or the Python installation is broken because they don't read all the way down to the bottom and see that it's a permission problem.

On the other hand, this doesn’t help the developer to find the cause of the failure. For this purpose, the DISTUTILS_DEBUG environment variable can be set to anything except an empty string, and distutils will now print detailed information about what it is doing, dump the full traceback when an exception occurs, and print the whole command line when an external program (like a C compiler) fails.
1. An Introduction to Distutils

This document covers using the Distutils to distribute your Python modules, concentrating on the role of developer/distributor: if you're looking for information on installing Python modules, you should refer to the Installing Python Modules (Legacy version) chapter.
1.1. Concepts & Terminology

Using the Distutils is quite simple, both for module developers and for users/administrators installing third-party modules. As a developer, your responsibilities (apart from writing solid, well-documented and well-tested code, of course!) are:

- write a setup script (`setup.py` by convention)
- (optional) write a setup configuration file
- create a source distribution
- (optional) create one or more built (binary) distributions

Each of these tasks is covered in this document.

Not all module developers have access to a multitude of platforms, so it's not always feasible to expect them to create a multitude of built distributions. It is hoped that a class of intermediaries, called packagers, will arise to address this need. Packagers will take source distributions released by module developers, build them on one or more platforms, and release the resulting built distributions. Thus, users on the most popular platforms will be able to install most popular Python module distributions in the most natural way for their platform, without having to run a single setup script or compile a line of code.
1.2. A Simple Example

The setup script is usually quite simple, although since it's written in Python, there are no arbitrary limits to what you can do with it, though you should be careful about putting arbitrarily expensive operations in your setup script. Unlike, say, Autoconf-style configure scripts, the setup script may be run multiple times in the course of building and installing your module distribution.

If all you want to do is distribute a module called foo, contained in a file foo.py, then your setup script can be as simple as this:

```python
from distutils.core import setup
setup(
    name='foo',
    version='1.0',
    py_modules=['foo'],
)
```

Some observations:

- most information that you supply to the Distutils is supplied as keyword arguments to the setup() function
- those keyword arguments fall into two categories: package metadata (name, version number) and information about what's in the package (a list of pure Python modules, in this case)
- modules are specified by module name, not filename (the same will hold true for packages and extensions)
- it's recommended that you supply a little more metadata, in particular your name, email address and a URL for the project (see section Writing the Setup Script for an example)

To create a source distribution for this module, you would create a setup script, setup.py, containing the above code, and run this command from a terminal:
python setup.py sdist

For Windows, open a command prompt windows (Start ➤ Accessories) and change the command to:

    setup.py sdist

`sdist` will create an archive file (e.g., tarball on Unix, ZIP file on Windows) containing your setup script `setup.py`, and your module `foo.py`. The archive file will be named `foo-1.0.tar.gz` (or `.zip`), and will unpack into a directory `foo-1.0`.

If an end-user wishes to install your `foo` module, all she has to do is download `foo-1.0.tar.gz` (or `.zip`), unpack it, and—from the `foo-1.0` directory—run

    python setup.py install

which will ultimately copy `foo.py` to the appropriate directory for third-party modules in their Python installation.

This simple example demonstrates some fundamental concepts of the Distutils. First, both developers and installers have the same basic user interface, i.e. the setup script. The difference is which Distutils `commands` they use: the `sdist` command is almost exclusively for module developers, while `install` is more often for installers (although most developers will want to install their own code occasionally).

If you want to make things really easy for your users, you can create one or more built distributions for them. For instance, if you are running on a Windows machine, and want to make things easy for other Windows users, you can create an executable installer (the most appropriate type of built distribution for this platform) with the `bdist_wininst` command. For example:
```bash
python setup.py bdist_wininst
```

will create an executable installer, `foo-1.0.win32.exe`, in the current directory.

Other useful built distribution formats are RPM, implemented by the `bdist_rpm` command, Solaris `pkgtool (bdist/pkgtool)`, and HP-UX `swinstall (bdist/sdux)`. For example, the following command will create an RPM file called `foo-1.0.noarch.rpm`:

```bash
python setup.py bdist_rpm
```

(The `bdist_rpm` command uses the `rpm` executable, therefore this has to be run on an RPM-based system such as Red Hat Linux, SuSE Linux, or Mandrake Linux.)

You can find out what distribution formats are available at any time by running

```bash
python setup.py bdist --help-formats
```
1.3. General Python terminology

If you’re reading this document, you probably have a good idea of what modules, extensions, and so forth are. Nevertheless, just to be sure that everyone is operating from a common starting point, we offer the following glossary of common Python terms:

module
  the basic unit of code reusability in Python: a block of code imported by some other code. Three types of modules concern us here: pure Python modules, extension modules, and packages.

pure Python module
  a module written in Python and contained in a single .py file (and possibly associated .pyc and/or .pyo files). Sometimes referred to as a “pure module.”

extension module
  a module written in the low-level language of the Python implementation: C/C++ for Python, Java for Jython. Typically contained in a single dynamically loadable pre-compiled file, e.g. a shared object (.so) file for Python extensions on Unix, a DLL (given the .pyd extension) for Python extensions on Windows, or a Java class file for Jython extensions. (Note that currently, the Distutils only handles C/C++ extensions for Python.)

package
  a module that contains other modules; typically contained in a directory in the filesystem and distinguished from other directories by the presence of a file __init__.py.

root package
  the root of the hierarchy of packages. (This isn’t really a package, since it doesn’t have an __init__.py file. But we have to call it
something.) The vast majority of the standard library is in the root package, as are many small, standalone third-party modules that don’t belong to a larger module collection. Unlike regular packages, modules in the root package can be found in many directories: in fact, every directory listed in `sys.path` contributes modules to the root package.
1.4. Distutils-specific terminology

The following terms apply more specifically to the domain of distributing Python modules using the Distutils:

module distribution
   a collection of Python modules distributed together as a single downloadable resource and meant to be installed *en masse*. Examples of some well-known module distributions are Numeric Python, PyXML, PIL (the Python Imaging Library), or mxBase. (This would be called a *package*, except that term is already taken in the Python context: a single module distribution may contain zero, one, or many Python packages.)

pure module distribution
   a module distribution that contains only pure Python modules and packages. Sometimes referred to as a “pure distribution.”

non-pure module distribution
   a module distribution that contains at least one extension module. Sometimes referred to as a “non-pure distribution.”

distribution root
   the top-level directory of your source tree (or source distribution); the directory where `setup.py` exists. Generally `setup.py` will be run from this directory.
4. Creating a Source Distribution

As shown in section A Simple Example, you use the `sdist` command to create a source distribution. In the simplest case,

```
python setup.py sdist
```

(assuming you haven't specified any `sdist` options in the setup script or config file), `sdist` creates the archive of the default format for the current platform. The default format is a gzip'ed tar file (.tar.gz) on Unix, and ZIP file on Windows.

You can specify as many formats as you like using the `--formats` option, for example:

```
python setup.py sdist --formats=gztar,zip
```

to create a gzipped tarball and a zip file. The available formats are:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>zip</td>
<td>zip file (.zip)</td>
<td>(1),(3)</td>
</tr>
<tr>
<td>gztar</td>
<td>gzip'ed tar file (.tar.gz)</td>
<td>(2)</td>
</tr>
<tr>
<td>bztar</td>
<td>bzip2'ed tar file (.tar.bz2)</td>
<td></td>
</tr>
<tr>
<td>ztar</td>
<td>compressed tar file (.tar.z)</td>
<td>(4)</td>
</tr>
<tr>
<td>tar</td>
<td>tar file (.tar)</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. default on Windows
2. default on Unix
3. requires either external `zip` utility or `zipfile` module (part of the standard Python library since Python 1.6)
4. requires the `compress` program.
When using any tar format (gztar, bztar, ztar or tar) under Unix, you can specify the owner and group names that will be set for each member of the archive.

For example, if you want all files of the archive to be owned by root:

```bash
python setup.py sdist --owner=root --group=root
```
4.1. Specifying the files to distribute

If you don’t supply an explicit list of files (or instructions on how to generate one), the `sdist` command puts a minimal default set into the source distribution:

- all Python source files implied by the `py_modules` and `packages` options
- all C source files mentioned in the `ext_modules` or `libraries` options
- scripts identified by the `scripts` option See Installing Scripts.
- anything that looks like a test script: `test/test*.py` (currently, the Distutils don’t do anything with test scripts except include them in source distributions, but in the future there will be a standard for testing Python module distributions)
- `README.txt` (or `README`), `setup.py` (or whatever you called your setup script), and `setup.cfg`
- all files that matches the `package_data` metadata. See Installing Package Data.
- all files that matches the `data_files` metadata. See Installing Additional Files.

Sometimes this is enough, but usually you will want to specify additional files to distribute. The typical way to do this is to write a *manifest template*, called `MANIFEST.in` by default. The manifest template is just a list of instructions for how to generate your manifest file, `MANIFEST`, which is the exact list of files to include in your source distribution. The `sdist` command processes this template and generates a manifest based on its instructions and what it finds in the filesystem.

If you prefer to roll your own manifest file, the format is simple: one
filename per line, regular files (or symlinks to them) only. If you do supply your own MANIFEST, you must specify everything: the default set of files described above does not apply in this case.

*Changed in version 2.7:* An existing generated MANIFEST will be regenerated without sdist comparing its modification time to the one of MANIFEST.in or setup.py.

*Changed in version 2.7.1:* MANIFEST files start with a comment indicating they are generated. Files without this comment are not overwritten or removed.

*Changed in version 2.7.3:* sdist will read a MANIFEST file if no MANIFEST.in exists, like it did before 2.7.

See The MANIFEST.in template section for a syntax reference.
4.2. Manifest-related options

The normal course of operations for the `sdist` command is as follows:

- if the manifest file (MANIFEST by default) exists and the first line does not have a comment indicating it is generated from MANIFEST.in, then it is used as is, unaltered
- if the manifest file doesn’t exist or has been previously automatically generated, read `MANIFEST.in` and create the manifest
- if neither MANIFEST nor MANIFEST.in exist, create a manifest with just the default file set
- use the list of files now in MANIFEST (either just generated or read in) to create the source distribution archive(s)

There are a couple of options that modify this behaviour. First, use the `--no-defaults` and `--no-prune` to disable the standard “include” and “exclude” sets.

Second, you might just want to (re)generate the manifest, but not create a source distribution:

```
python setup.py sdist --manifest-only
```

-o is a shortcut for `--manifest-only`. 
4.3. The MANIFEST.in template

A `MANIFEST.in` file can be added in a project to define the list of files to include in the distribution built by the `sdist` command.

When `sdist` is run, it will look for the `MANIFEST.in` file and interpret it to generate the `MANIFEST` file that contains the list of files that will be included in the package.

This mechanism can be used when the default list of files is not enough. (See Specifying the files to distribute).

4.3.1. Principle

The manifest template has one command per line, where each command specifies a set of files to include or exclude from the source distribution. For an example, let's look at the Distutils' own manifest template:

```
include *.txt
recursive-include examples *.txt *.py
prune examples/sample?/build
```

The meanings should be fairly clear: include all files in the distribution root matching `*.txt`, all files anywhere under the `examples` directory matching `*.txt` or `*.py`, and exclude all directories matching `examples/sample?/build`. All of this is done after the standard include set, so you can exclude files from the standard set with explicit instructions in the manifest template. (Or, you can use the `--no-defaults` option to disable the standard set entirely.)

The order of commands in the manifest template matters: initially, we have the list of default files as described above, and each command
in the template adds to or removes from that list of files. Once we have fully processed the manifest template, we remove files that should not be included in the source distribution:

- all files in the Distutils “build” tree (default build/)
- all files in directories named RCS, CVS, .svn, .hg, .git, .bzr or _darcs

Now we have our complete list of files, which is written to the manifest for future reference, and then used to build the source distribution archive(s).

You can disable the default set of included files with the --no-defaults option, and you can disable the standard exclude set with --no-prune.

Following the Distutils’ own manifest template, let’s trace how the sdist command builds the list of files to include in the Distutils source distribution:

1. include all Python source files in the distutils and distutils/command subdirectories (because packages corresponding to those two directories were mentioned in the packages option in the setup script—see section Writing the Setup Script)
2. include README.txt, setup.py, and setup.cfg (standard files)
3. include test/test*.py (standard files)
4. include *.txt in the distribution root (this will find README.txt a second time, but such redundancies are weeded out later)
5. include anything matching *.txt or *.py in the sub-tree under examples,
6. exclude all files in the sub-trees starting at directories matching examples/sample?/build—this may exclude files included by the
previous two steps, so it's important that the `prune` command in the manifest template comes after the `recursive-include` command

7. exclude the entire build tree, and any `RCS`, `CVS`, `.svn`, `.hg`, `.git`, `.bzr` and `_darcs` directories

Just like in the setup script, file and directory names in the manifest template should always be slash-separated; the Distutils will take care of converting them to the standard representation on your platform. That way, the manifest template is portable across operating systems.

### 4.3.2. Commands

The manifest template commands are:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>include pat1 pat2 ...</code></td>
<td>include all files matching any of the listed patterns</td>
</tr>
<tr>
<td><code>exclude pat1 pat2 ...</code></td>
<td>exclude all files matching any of the listed patterns</td>
</tr>
<tr>
<td><code>recursive-include dir pat1 pat2 ...</code></td>
<td>include all files under <code>dir</code> matching any of the listed patterns</td>
</tr>
<tr>
<td><code>recursive-exclude dir pat1 pat2 ...</code></td>
<td>exclude all files under <code>dir</code> matching any of the listed patterns</td>
</tr>
<tr>
<td><code>global-include pat1 pat2 ...</code></td>
<td>include all files anywhere in the source tree matching — &amp; any of the listed patterns</td>
</tr>
<tr>
<td><code>global-exclude pat1 pat2 ...</code></td>
<td>exclude all files anywhere in the source tree matching — &amp; any of the listed patterns</td>
</tr>
<tr>
<td><code>prune dir</code></td>
<td>exclude all files under <code>dir</code></td>
</tr>
<tr>
<td><code>graft dir</code></td>
<td>include all files under <code>dir</code></td>
</tr>
</tbody>
</table>
The patterns here are Unix-style “glob” patterns: * matches any sequence of regular filename characters, ? matches any single regular filename character, and [range] matches any of the characters in range (e.g., a-z, a-zA-Z, a-f0-9_). The definition of “regular filename character” is platform-specific: on Unix it is anything except slash; on Windows anything except backslash or colon.
5. Creating Built Distributions

A “built distribution” is what you’re probably used to thinking of either as a “binary package” or an “installer” (depending on your background). It’s not necessarily binary, though, because it might contain only Python source code and/or byte-code; and we don’t call it a package, because that word is already spoken for in Python. (And “installer” is a term specific to the world of mainstream desktop systems.)

A built distribution is how you make life as easy as possible for installers of your module distribution: for users of RPM-based Linux systems, it’s a binary RPM; for Windows users, it’s an executable installer; for Debian-based Linux users, it’s a Debian package; and so forth. Obviously, no one person will be able to create built distributions for every platform under the sun, so the Distutils are designed to enable module developers to concentrate on their specialty—writing code and creating source distributions—while an intermediary species called packagers springs up to turn source distributions into built distributions for as many platforms as there are packagers.

Of course, the module developer could be his own packager; or the packager could be a volunteer “out there” somewhere who has access to a platform which the original developer does not; or it could be software periodically grabbing new source distributions and turning them into built distributions for as many platforms as the software has access to. Regardless of who they are, a packager uses the setup script and the bdist command family to generate built distributions.

As a simple example, if I run the following command in the Distutils source tree:
then the Distutils builds my module distribution (the Distutils itself in this case), does a “fake” installation (also in the build directory), and creates the default type of built distribution for my platform. The default format for built distributions is a “dumb” tar file on Unix, and a simple executable installer on Windows. (That tar file is considered “dumb” because it has to be unpacked in a specific location to work.)

Thus, the above command on a Unix system creates Distutils-1.0.plat.tar.gz; unpacking this tarball from the right place installs the Distutils just as though you had downloaded the source distribution and run python setup.py install. (The “right place” is either the root of the filesystem or Python’s prefix directory, depending on the options given to the bdist_dumb command; the default is to make dumb distributions relative to prefix.)

Obviously, for pure Python distributions, this isn’t any simpler than just running python setup.py install—but for non-pure distributions, which include extensions that would need to be compiled, it can mean the difference between someone being able to use your extensions or not. And creating “smart” built distributions, such as an RPM package or an executable installer for Windows, is far more convenient for users even if your distribution doesn’t include any extensions.

The bdist command has a --formats option, similar to the sdist command, which you can use to select the types of built distribution to generate: for example,

```
python setup.py bdist --format=zip
```

would, when run on a Unix system, create Distutils-1.0.plat.zip—again, this archive would be unpacked from the root directory to
install the Distutils.

The available formats for built distributions are:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>gztar</td>
<td>gzipped tar file (.tar.gz)</td>
<td>(1),(3)</td>
</tr>
<tr>
<td>ztar</td>
<td>compressed tar file (.tar.Z)</td>
<td>(3)</td>
</tr>
<tr>
<td>tar</td>
<td>tar file (.tar)</td>
<td>(3)</td>
</tr>
<tr>
<td>zip</td>
<td>zip file (.zip)</td>
<td>(2),(4)</td>
</tr>
<tr>
<td>rpm</td>
<td>RPM</td>
<td>(5)</td>
</tr>
<tr>
<td>pkgtool</td>
<td>Solaris pkgtool</td>
<td></td>
</tr>
<tr>
<td>sdux</td>
<td>HP-UX swinstall</td>
<td></td>
</tr>
<tr>
<td>wininst</td>
<td>self-extracting ZIP file for Windows</td>
<td>(4)</td>
</tr>
<tr>
<td>msi</td>
<td>Microsoft Installer.</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. default on Unix
2. default on Windows
3. requires external utilities: tar and possibly one of gzip, bzip2, or compress
4. requires either external zip utility or zipfile module (part of the standard Python library since Python 1.6)
5. requires external rpm utility, version 3.0.4 or better (use rpm --version to find out which version you have)

You don’t have to use the bdist command with the --formats option; you can also use the command that directly implements the format you’re interested in. Some of these bdist “sub-commands” actually generate several similar formats; for instance, the bdist_dumb command generates all the “dumb” archive formats (tar, ztar, gztar, and zip), and bdist_rpm generates both binary and source RPMs. The bdist sub-commands, and the formats generated by each, are:
<table>
<thead>
<tr>
<th>Command</th>
<th>Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>bdist_dumb</td>
<td>tar, ztar, gztar, zip</td>
</tr>
<tr>
<td>bdist_rpm</td>
<td>rpm, srpm</td>
</tr>
<tr>
<td>bdist_wininst</td>
<td>wininst</td>
</tr>
<tr>
<td>bdist_msi</td>
<td>msi</td>
</tr>
</tbody>
</table>

The following sections give details on the individual `bdist_*` commands.
5.1. Creating dumb built distributions
5.2. Creating RPM packages

The RPM format is used by many popular Linux distributions, including Red Hat, SuSE, and Mandrake. If one of these (or any of the other RPM-based Linux distributions) is your usual environment, creating RPM packages for other users of that same distribution is trivial. Depending on the complexity of your module distribution and differences between Linux distributions, you may also be able to create RPMs that work on different RPM-based distributions.

The usual way to create an RPM of your module distribution is to run the **bdist_rpm** command:

```
python setup.py bdist_rpm
```

or the **bdist** command with the **--format** option:

```
python setup.py bdist --format=rpm
```

The former allows you to specify RPM-specific options; the latter allows you to easily specify multiple formats in one run. If you need to do both, you can explicitly specify multiple **bdist_*** commands and their options:

```
python setup.py bdist_rpm --packager="John Doe <jdoe@example.org>
bdist_wininst --target-version="2.0"
```

Creating RPM packages is driven by a `.spec` file, much as using the Distutils is driven by the setup script. To make your life easier, the **bdist_rpm** command normally creates a `.spec` file based on the information you supply in the setup script, on the command line, and in any Distutils configuration files. Various options and sections in the `.spec` file are derived from options in the setup script as follows:
<table>
<thead>
<tr>
<th>RPM <code>.spec</code> file option or section</th>
<th>Distutils setup script option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>name</td>
</tr>
<tr>
<td>Summary (in preamble)</td>
<td>description</td>
</tr>
<tr>
<td>Version</td>
<td>version</td>
</tr>
<tr>
<td>Vendor</td>
<td>author and author_email, or — &amp; maintainer and maintainer_email</td>
</tr>
<tr>
<td>Copyright</td>
<td>license</td>
</tr>
<tr>
<td>Url</td>
<td>url</td>
</tr>
<tr>
<td>%description (section)</td>
<td>long_description</td>
</tr>
</tbody>
</table>

Additionally, there are many options in `.spec` files that don’t have corresponding options in the setup script. Most of these are handled through options to the `bdist_rpm` command as follows:

<table>
<thead>
<tr>
<th>RPM <code>.spec</code> file option or section</th>
<th><code>bdist_rpm</code> option</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release</td>
<td>release</td>
<td>“1”</td>
</tr>
<tr>
<td>Group</td>
<td>group</td>
<td>“Development/Libraries”</td>
</tr>
<tr>
<td>Vendor</td>
<td>vendor</td>
<td>(see above)</td>
</tr>
<tr>
<td>Packager</td>
<td>packager</td>
<td>(none)</td>
</tr>
<tr>
<td>Provides</td>
<td>provides</td>
<td>(none)</td>
</tr>
<tr>
<td>Requires</td>
<td>requires</td>
<td>(none)</td>
</tr>
<tr>
<td>Conflicts</td>
<td>conflicts</td>
<td>(none)</td>
</tr>
<tr>
<td>Obsoletes</td>
<td>obsoletes</td>
<td>(none)</td>
</tr>
<tr>
<td>Distribution</td>
<td>distribution_name</td>
<td>(none)</td>
</tr>
<tr>
<td>BuildRequires</td>
<td>build_requires</td>
<td>(none)</td>
</tr>
<tr>
<td>Icon</td>
<td>icon</td>
<td>(none)</td>
</tr>
</tbody>
</table>

Obviously, supplying even a few of these options on the command-line would be tedious and error-prone, so it’s usually best to put them in the setup configuration file, `setup.cfg`—see section Writing the Setup Configuration File. If you distribute or package many Python
module distributions, you might want to put options that apply to all of them in your personal Distutils configuration file (~/.pydistutils.cfg). If you want to temporarily disable this file, you can pass the --no-user-cfg option to setup.py.

There are three steps to building a binary RPM package, all of which are handled automatically by the Distutils:

1. create a .spec file, which describes the package (analogous to the Distutils setup script; in fact, much of the information in the setup script winds up in the .spec file)
2. create the source RPM
3. create the “binary” RPM (which may or may not contain binary code, depending on whether your module distribution contains Python extensions)

Normally, RPM bundles the last two steps together; when you use the Distutils, all three steps are typically bundled together.

If you wish, you can separate these three steps. You can use the --spec-only option to make bdist_rpm just create the .spec file and exit; in this case, the .spec file will be written to the “distribution directory”—normally dist/, but customizable with the --dist-dir option. (Normally, the .spec file winds up deep in the “build tree,” in a temporary directory created by bdist_rpm.)
5.3. Creating Windows Installers

Executable installers are the natural format for binary distributions on Windows. They display a nice graphical user interface, display some information about the module distribution to be installed taken from the metadata in the setup script, let the user select a few options, and start or cancel the installation.

Since the metadata is taken from the setup script, creating Windows installers is usually as easy as running:

```
python setup.py bdist_wininst
```

or the `bdist` command with the `--formats` option:

```
python setup.py bdist --formats=wininst
```

If you have a pure module distribution (only containing pure Python modules and packages), the resulting installer will be version independent and have a name like `foo-1.0.win32.exe`. These installers can even be created on Unix platforms or Mac OS X.

If you have a non-pure distribution, the extensions can only be created on a Windows platform, and will be Python version dependent. The installer filename will reflect this and now has the form `foo-1.0.win32-py2.0.exe`. You have to create a separate installer for every Python version you want to support.

The installer will try to compile pure modules into bytecode after installation on the target system in normal and optimizing mode. If you don't want this to happen for some reason, you can run the `bdist_wininst` command with the `--no-target-compile` and/or the `--no-target-optimize` option.
By default the installer will display the cool “Python Powered” logo when it is run, but you can also supply your own 152x261 bitmap which must be a Windows .bmp file with the --bitmap option.

The installer will also display a large title on the desktop background window when it is run, which is constructed from the name of your distribution and the version number. This can be changed to another text by using the --title option.

The installer file will be written to the “distribution directory” — normally dist/, but customizable with the --dist-dir option.
5.4. Cross-compiling on Windows

Starting with Python 2.6, distutils is capable of cross-compiling between Windows platforms. In practice, this means that with the correct tools installed, you can use a 32bit version of Windows to create 64bit extensions and vice-versa.

To build for an alternate platform, specify the `--plat-name` option to the build command. Valid values are currently ‘win32’, ‘win-amd64’ and ‘win-ia64’. For example, on a 32bit version of Windows, you could execute:

```bash
python setup.py build --plat-name=win-amd64
```

to build a 64bit version of your extension. The Windows Installers also support this option, so the command:

```bash
python setup.py build --plat-name=win-amd64 bdist_wininst
```

would create a 64bit installation executable on your 32bit version of Windows.

To cross-compile, you must download the Python source code and cross-compile Python itself for the platform you are targeting - it is not possible from a binary installation of Python (as the .lib etc file for other platforms are not included.) In practice, this means the user of a 32 bit operating system will need to use Visual Studio 2008 to open the `PCBuild/PCbuild.sln` solution in the Python source tree and build the “x64” configuration of the ‘pythoncore’ project before cross-compiling extensions is possible.

Note that by default, Visual Studio 2008 does not install 64bit compilers or tools. You may need to reexecute the Visual Studio setup process and select these tools (using Control Panel->
5.4.1. The Postinstallation script

Starting with Python 2.3, a postinstallation script can be specified with the \texttt{--install-script} option. The basename of the script must be specified, and the script filename must also be listed in the scripts argument to the setup function.

This script will be run at installation time on the target system after all the files have been copied, with \texttt{argv[1]} set to \texttt{-install}, and again at uninstallation time before the files are removed with \texttt{argv[1]} set to \texttt{-remove}.

The installation script runs embedded in the windows installer, every output (\texttt{sys.stdout}, \texttt{sys.stderr}) is redirected into a buffer and will be displayed in the GUI after the script has finished.

Some functions especially useful in this context are available as additional built-in functions in the installation script.

\texttt{directory_created(path)}
\texttt{file_created(path)}

These functions should be called when a directory or file is created by the postinstall script at installation time. It will register \texttt{path} with the uninstaller, so that it will be removed when the distribution is uninstalled. To be safe, directories are only removed if they are empty.

\texttt{get_special_folder_path(csidl_string)}

This function can be used to retrieve special folder locations on Windows like the Start Menu or the Desktop. It returns the full path to the folder. \texttt{csidl_string} must be one of the following strings:
If the folder cannot be retrieved, `OSError` is raised.

Which folders are available depends on the exact Windows version, and probably also the configuration. For details refer to Microsoft's documentation of the `SHGetSpecialFolderPath()` function.

```python
createShortcut(target, description, filename[, arguments[, workdir[, iconpath[, iconindex]]]])
```

This function creates a shortcut. `target` is the path to the program to be started by the shortcut. `description` is the description of the shortcut. `filename` is the title of the shortcut that the user will see. `arguments` specifies the command line arguments, if any. `workdir` is the working directory for the program. `iconpath` is the file containing the icon for the shortcut, and `iconindex` is the index of the icon in the file `iconpath`. Again, for details consult the Microsoft documentation for the IShellLink interface.
5.5. Vista User Access Control (UAC)

Starting with Python 2.6, bdist_wininst supports a `--user-access-control` option. The default is `none` (meaning no UAC handling is done), and other valid values are `auto` (meaning prompt for UAC elevation if Python was installed for all users) and `force` (meaning always prompt for elevation).
6. The Python Package Index (PyPI)

The Python Package Index (PyPI) stores meta-data describing distributions packaged with distutils, as well as package data like distribution files if a package author wishes.

Distutils provides the register and upload commands for pushing meta-data and distribution files to PyPI, respectively. See Distutils commands for information on these commands.
6.1. PyPI overview

PyPI lets you submit any number of versions of your distribution to the index. If you alter the meta-data for a particular version, you can submit it again and the index will be updated.

PyPI holds a record for each (name, version) combination submitted. The first user to submit information for a given name is designated the Owner of that name. Changes can be submitted through the register command or through the web interface. Owners can designate other users as Owners or Maintainers. Maintainers can edit the package information, but not designate new Owners or Maintainers.

By default PyPI displays only the newest version of a given package. The web interface lets one change this default behavior and manually select which versions to display and hide.

For each version, PyPI displays a home page. The home page is created from the long_description which can be submitted via the register command. See PyPI package display for more information.
6.2. Distutils commands

Distutils exposes two commands for submitting package data to PyPI: the register command for submitting meta-data to PyPI and the upload command for submitting distribution files. Both commands read configuration data from a special file called a .pypirc file.

6.2.1. The register command

The distutils command register is used to submit your distribution’s meta-data to an index server. It is invoked as follows:

```python
python setup.py register
```

Distutils will respond with the following prompt:

```
running register
We need to know who you are, so please choose either:
  1. use your existing login,
  2. register as a new user,
  3. have the server generate a new password for you (and email)
  4. quit
Your selection [default 1]:
```

Note: if your username and password are saved locally, you will not see this menu. Also, refer to The .pypirc file for how to store your credentials in a .pypirc file.

If you have not registered with PyPI, then you will need to do so now. You should choose option 2, and enter your details as required. Soon after submitting your details, you will receive an email which will be used to confirm your registration.

Once you are registered, you may choose option 1 from the menu. You will be prompted for your PyPI username and password, and
register will then submit your meta-data to the index.

See Additional command options for options to the register command.

6.2.2. The upload command

New in version 2.5.

The distutils command upload pushes the distribution files to PyPI.

The command is invoked immediately after building one or more distribution files. For example, the command

```
python setup.py sdist bdist_wininst upload
```

will cause the source distribution and the Windows installer to be uploaded to PyPI. Note that these will be uploaded even if they are built using an earlier invocation of setup.py, but that only distributions named on the command line for the invocation including the upload command are uploaded.

If a register command was previously called in the same command, and if the password was entered in the prompt, upload will reuse the entered password. This is useful if you do not want to store a password in clear text in a .pypirc file.

You can use the --sign option to tell upload to sign each uploaded file using GPG (GNU Privacy Guard). The gpg program must be available for execution on the system PATH. You can also specify which key to use for signing using the --identity=name option.

See Additional command options for additional options to the upload command.
6.2.3. Additional command options

This section describes options common to both the register and upload commands.

The --repository or -r option lets you specify a PyPI server different from the default. For example:

```
python setup.py sdist bdist_wininst upload -r https://example.com/
```

For convenience, a name can be used in place of the URL when the .pypirc file is configured to do so. For example:

```
python setup.py register -r other
```

See The .pypirc file for more information on defining alternate servers.

The --show-response option displays the full response text from the PyPI server, which is useful when debugging problems with registering and uploading.

6.2.4. The .pypirc file

The register and upload commands both check for the existence of a .pypirc file at the location $HOME/.pypirc. If this file exists, the command uses the username, password, and repository URL configured in the file. The format of a .pypirc file is as follows:

```
[distutils]
index-servers =
  pypi

[pypi]
repository: <repository-url>
```
The *distutils* section defines an *index-servers* variable that lists the name of all sections describing a repository.

Each section describing a repository defines three variables:

- *repository*, that defines the url of the PyPI server. Defaults to https://www.python.org/pypi.

- *username*, which is the registered username on the PyPI server.

- *password*, that will be used to authenticate. If omitted the user will be prompt to type it when needed.

If you want to define another server a new section can be created and listed in the *index-servers* variable:

```bash
[distutils]
index-servers =
    pypi
    other

[pypi]
repository: <repository-url>
username: <username>
password: <password>

[other]
repository: https://example.com/pypi
username: <username>
password: <password>
```

This allows the *register* and *upload* commands to be called with the *--repository* option as described in Additional command options.

Specifically, you might want to add the PyPI Test Repository to your `.pypirc` to facilitate testing before doing your first upload to PyPI.
itself.
6.3. PyPI package display

The `long_description` field plays a special role at PyPI. It is used by the server to display a home page for the registered package.

If you use the `reStructuredText` syntax for this field, PyPI will parse it and display an HTML output for the package home page.

The `long_description` field can be attached to a text file located in the package:

```python
from distutils.core import setup

with open('README.txt') as file:
    long_description = file.read()

setup(name='Distutils',
      long_description=long_description)
```

In that case, `README.txt` is a regular `reStructuredText` text file located in the root of the package besides `setup.py`.

To prevent registering broken `reStructuredText` content, you can use the `rst2html` program that is provided by the `docutils` package and check the `long_description` from the command line:

```bash
$ python setup.py --long-description | rst2html.py > output.html
```

`docutils` will display a warning if there’s something wrong with your syntax. Because PyPI applies additional checks (e.g. by passing `--no-raw` to `rst2html.py` in the command above), being able to run the command above without warnings does not guarantee that PyPI will convert the content successfully.
7. Examples

This chapter provides a number of basic examples to help get started with distutils. Additional information about using distutils can be found in the Distutils Cookbook.

See also:

Distutils Cookbook
Collection of recipes showing how to achieve more control over distutils.
If you're just distributing a couple of modules, especially if they don't live in a particular package, you can specify them individually using the `py_modules` option in the setup script.

In the simplest case, you'll have two files to worry about: a setup script and the single module you're distributing, `foo.py` in this example:

```
<root>/
    setup.py
    foo.py
```

(In all diagrams in this section, `<root>` will refer to the distribution root directory.) A minimal setup script to describe this situation would be:

```
from distutils.core import setup
setup(name='foo',
    version='1.0',
    py_modules=['foo'],
)
```

Note that the name of the distribution is specified independently with the `name` option, and there's no rule that says it has to be the same as the name of the sole module in the distribution (although that's probably a good convention to follow). However, the distribution name is used to generate filenames, so you should stick to letters, digits, underscores, and hyphens.

Since `py_modules` is a list, you can of course specify multiple modules, eg. if you're distributing modules `foo` and `bar`, your setup might look like this:

```
<root>/
```
and the setup script might be

```python
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      py_modules=['foo', 'bar'],
)
```

You can put module source files into another directory, but if you have enough modules to do that, it's probably easier to specify modules by package rather than listing them individually.
7.2. Pure Python distribution (by package)

If you have more than a couple of modules to distribute, especially if they are in multiple packages, it’s probably easier to specify whole packages rather than individual modules. This works even if your modules are not in a package; you can just tell the Distutils to process modules from the root package, and that works the same as any other package (except that you don’t have to have an \_init\_.py file).

The setup script from the last example could also be written as

```python
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      packages=[''],
)
```

(The empty string stands for the root package.)

If those two files are moved into a subdirectory, but remain in the root package, e.g.:

```
<root>/
  setup.py
  src/
     foo.py
     bar.py
```

then you would still specify the root package, but you have to tell the Distutils where source files in the root package live:

```python
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      package_dir={'': 'src'},
      packages=[''],
)
```
More typically, though, you will want to distribute multiple modules in the same package (or in sub-packages). For example, if the foo and bar modules belong in package foobar, one way to layout your source tree is

```python
<root/>
    setup.py
    foobar/
        __init__.py
        foo.py
        bar.py
```

This is in fact the default layout expected by the Distutils, and the one that requires the least work to describe in your setup script:

```python
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      packages=['foobar'],
)
```

If you want to put modules in directories not named for their package, then you need to use the `package_dir` option again. For example, if the src directory holds modules in the foobar package:

```python
<root/>
    setup.py
    src/
        __init__.py
        foo.py
        bar.py
```

an appropriate setup script would be

```python
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      package_dir={'foobar': 'src'},
      packages=['foobar'],
)
```
Or, you might put modules from your main package right in the distribution root:

```
<root/>

    setup.py
    __init__.py
    foo.py
    bar.py
```

in which case your setup script would be

```
from distutils.core import setup
setup(name='foobar',
       version='1.0',
       package_dir={'foobar': ''},
       packages=['foobar'],
)
```

(The empty string also stands for the current directory.)

If you have sub-packages, they must be explicitly listed in `packages`, but any entries in `package_dir` automatically extend to sub-packages. (In other words, the Distutils does *not* scan your source tree, trying to figure out which directories correspond to Python packages by looking for `__init__.py` files.) Thus, if the default layout grows a sub-package:

```
<root/>

    setup.py
    foobar/

          __init__.py
    foo.py
    bar.py
        subfoo/  
                  __init__.py
    blah.py
```

then the corresponding setup script would be

```
from distutils.core import setup
```
setup(name='foobar',
    version='1.0',
    packages=['foobar', 'foobar.subfoo'],
)
7.3. Single extension module

Extension modules are specified using the `ext_modules` option. `package_dir` has no effect on where extension source files are found; it only affects the source for pure Python modules. The simplest case, a single extension module in a single C source file, is:

```python
<root>/
    setup.py
    foo.c
```

If the `foo` extension belongs in the root package, the setup script for this could be

```python
from distutils.core import setup
from distutils.extension import Extension
setup(name='foobar',
      version='1.0',
      ext_modules=[Extension('foo', ['foo.c'])],
)
```

If the extension actually belongs in a package, say `foopkg`, then

With exactly the same source tree layout, this extension can be put in the `foopkg` package simply by changing the name of the extension:

```python
from distutils.core import setup
from distutils.extension import Extension
setup(name='foobar',
      version='1.0',
      ext_modules=[Extension('foopkg.foo', ['foo.c'])],
)
```